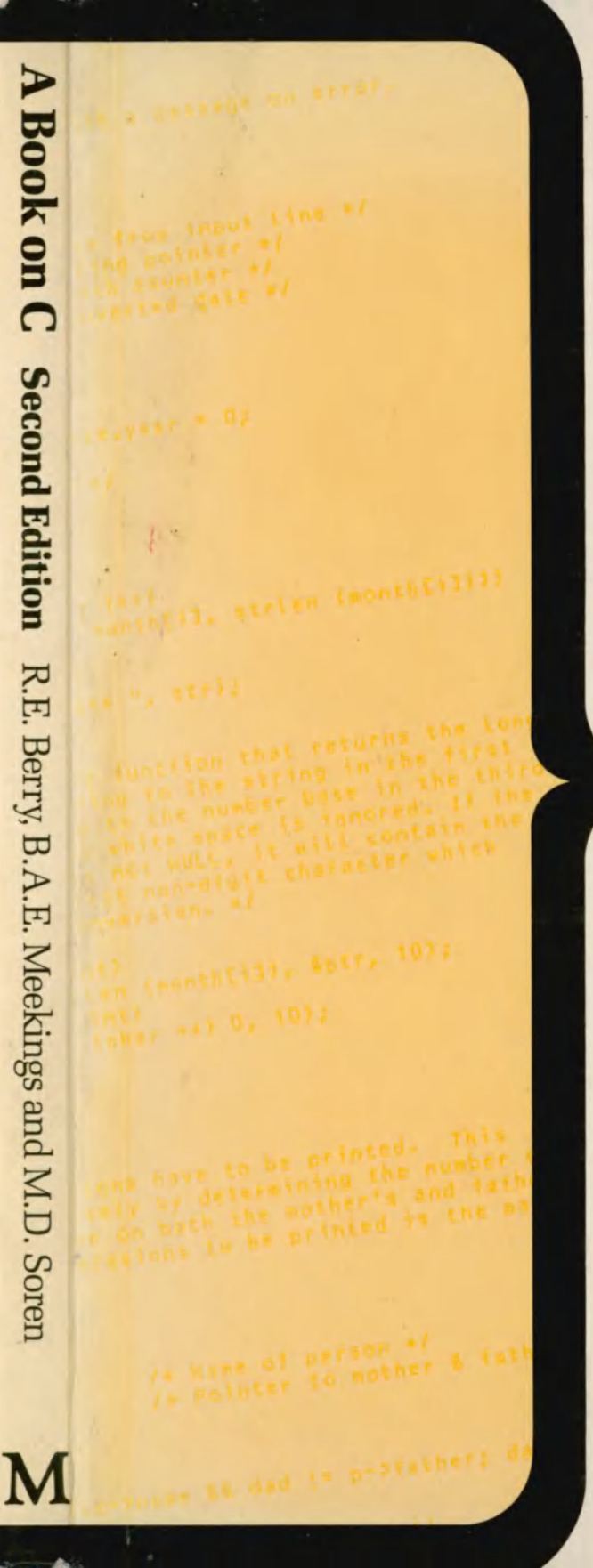


A Book on C

R.E. Berry,
B.A.E. Meekings
and M.D. Soren

Second Edition

A Book on C Second Edition R.E. Berry, B.A.E. Meekings and M.D. Soren

M

Macmillan Computer Science Series

Consulting Editor
Professor F. H. Sumner, University of Manchester

S. T. Allworth and R. N. Zobel, *Introduction to Real-time Software Design, second edition*
Ian O. Angell and Gareth Griffith, *High-resolution Computer Graphics Using FORTRAN 77*
Ian O. Angell and Gareth Griffith, *High-resolution Computer Graphics Using Pascal*
M. A. Azmoodeh, *Abstract Data Types and Algorithms*
C. Bamford and P. Curran, *Data Structures, Files and Databases*
Philip Barker, *Author Languages for CAL*
A. N. Barrett and A. L. Mackay, *Spatial Structure and the Microcomputer*
R. E. Berry, B. A. E. Meekings and M. D. Soren, *A Book on C, second edition*
G. M. Birtwistle, *Discrete Event Modelling on Simula*
T. B. Boffey, *Graph Theory in Operations Research*
Richard Bornat, *Understanding and Writing Compilers*
Linda E. M. Brackenbury, *Design of VLSI Systems – A Practical Introduction*
J. K. Buckle, *Software Configuration Management*
W. D. Burnham and A. R. Hall, *Prolog Programming and Applications*
J. C. Cluley, *Interfacing to Microprocessors*
Robert Cole, *Computer Communications, second edition*
Derek Coleman, *A Structured Programming Approach to Data*
Andrew J. T. Colin, *Fundamentals of Computer Science*
Andrew J. T. Colin, *Programming and Problem-solving in Algol 68*
S. M. Deen, *Fundamentals of Data Base Systems*
S. M. Deen, *Principles and Practice of Database Systems*
Tim Denvir, *Introduction to Discrete Mathematics for Software Engineering*
P. M. Dew and K. R. James, *Introduction to Numerical Computation in Pascal*
M. R. M. Dunsmuir and G. J. Davies, *Programming the UNIX System*
K. C. E. Gee, *Introduction to Local Area Computer Networks*
J. B. Gosling, *Design of Arithmetic Units for Digital Computers*
Roger Hutty, *Fortran for Students*
Roger Hutty, *Z80 Assembly Language Programming for Students*
Roland N. Ibbett, *The Architecture of High Performance Computers*
Patrick Jaulent, *The 68000 – Hardware and Software*
P. Jaulent, L. Baticle and P. Pillot, *68020-68030 Microprocessors and their Coprocessors*
J. M. King and J. P. Pardoe, *Program Design Using JSP – A Practical Introduction*
H. Kopetz, *Software Reliability*
E. V. Krishnamurthy, *Introductory Theory of Computer Science*
V. P. Lane, *Security of Computer Based Information Systems*

(*continued overleaf*)

Graham Lee, *From Hardware to Software – an introduction to computers*
A. M. Lister, *Fundamentals of Operating Systems, third edition*
G. P. McKeown and V. J. Rayward-Smith, *Mathematics for Computing*
Brian Meek, *Fortran, PL/1 and the Algols*
A. Mével and T. Guéguen, *Smalltalk-80*
Barry Morrell and Peter Whittle, *CP/M 80 Programmer's Guide*
Derrick Morris, *System Programming Based on the PDP11*
Pim Oets, *MS-DOS and PC-DOS – A Practical Guide*
Christian Queinnec, *LISP*
E. R. Redfern, *Introduction to Pascal for Computational Mathematics*
Gordon Reece, *Microcomputer Modelling by Finite Differences*
W. P. Salman, O. Tisserand and B. Toulout, *FORTH*
L. E. Scales, *Introduction to Non-linear Optimization*
Peter S. Sell, *Expert Systems – A Practical Introduction*
Colin J. Theaker and Graham R. Brookes, *A Practical Course on Operating Systems*
J-M. Trio, *8086-8088 Architecture and Programming*
M. J. Usher, *Information Theory for Information Technologists*
B. S. Walker, *Understanding Microprocessors*
Peter J. L. Wallis, *Portable Programming*
Colin Walls, *Programming Dedicated Microprocessors*
I. R. Wilson and A. M. Addyman, *A Practical Introduction to Pascal – with BS6192, second edition*

Non-series

Roy Anderson, *Management, Information Systems and Computers*
J. E. Bingham and G. W. P. Davies, *A Handbook of Systems Analysis, second edition*
J. E. Bingham and G. W. P. Davies, *Planning for Data Communications*
N. Frude, *A Guide to SPSS/PC+*

A Book on C

R. E. Berry
B.A.E. Meekings
and
M. D. Soren

Second Edition

MACMILLAN
EDUCATION

© R. E. Berry and B. A. E. Meekings 1984
© R. E. Berry, B. A. E. Meekings and M. D. Soren 1988

All rights reserved. No reproduction, copy or transmission of this publication may be made without written permission.

No paragraph of this publication may be reproduced, copied or transmitted save with written permission or in accordance with the provisions of the Copyright Act 1956 (as amended), or under the terms of any licence permitting limited copying issued by the Copyright Licensing Agency, 33-4 Alfred Place, London WC1E 7DP.

Any person who does any unauthorised act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

First edition 1984
Reprinted 1985, 1986
Second edition 1988

Published by
MACMILLAN EDUCATION LTD
Houndmills, Basingstoke, Hampshire RG21 2XS
and London
Companies and representatives
throughout the world

Printed in Hong Kong

British Library Cataloguing in Publication Data
Berry, Robert E.
A book on C. – 2nd ed. – (Macmillan computer science series).
1. C (Computer program language)
I. Title II. Meekings, B. A. E.
III. Soren, M. D.
005.13'3 QA76.73.C15

ISBN 0-333-46529-6

For
Marion, Judy and Paul,
Toby, Tim, Lucy, and Ben,
Patrick

Contents

Preface to the Second Edition *ix*

Acknowledgements *x*

Introduction 1

1 Program Structure **5**
2 Functions **14**
3 Output and Input **26**
4 Decisions **38**
5 Loops **45**
6 Operators **54**
7 Arrays **65**
8 More Data Types **76**
9 Pointers Revisited **84**
10 The C Preprocessor **99**
11 Programming Style **105**

Appendix 1: RatC 117
Appendix 2: RatC Listing 130
Appendix 3: C Style Analysis 186
Appendix 4: Screen Characteristics 199
Appendix 5: Tabulated and Listed Information 201
Alphabetic list of C reserved words 201
C arithmetic operators 202
C basic data types 203
ASCII character set (octal) 204
ASCII character set (decimal) 205
Escape characters 206
Conversion characters for output 206
Conversion characters for input 207
Input-output functions 208
Functions for string operations 210
Mathematical functions 211

Appendix 6: Syntax Diagrams for C 213

References 229

Index 231

Preface to the Second Edition

When we originally wrote this book, it was with the intention of providing an introduction to a powerful and complex programming language. Its power is amply demonstrated by its use to code a significant portion of the Unix operating system, its complexity by the necessity for books like Feuer's *The C Puzzle Book*. We deliberately omitted some of the more complex features of the language, believing that their description would easily warrant another entire book.

Since the first edition was published, Bob Berry died tragically. He was what I like to call a true 'software practitioner', well versed in every aspect of computer software – from research to education to practical application. He was loved and respected by his students, colleagues and friends alike, and is sorely missed.

Mike Soren is (among other things) a veteran C programmer and has stepped into the breach to make possible the other difference between this edition and the first – the expansion of the text to include all the features of C that were previously left out. This does not represent a change of intent – it is still our belief that no single book can teach anyone to program effectively. That comes only with experience.

At least you now will have the tools to become not just a good, but a great, programmer.

December 1987 Brian Meekings

Acknowledgements

Producing and collecting the material for a book is a time-consuming activity even when shared. The help of others in the activity is always greatly appreciated. It is our pleasure to record sincere thanks to those people whose help has been especially valuable, notably Peter Hurley, Chi Yip, and Jerry Hall. While many of their helpful comments and suggestions have found their way into the text and supporting software, the responsibility for the accuracy and integrity of the text rests solely with the authors.

March 1984

Bob Berry
Brian Meekings

Introduction

Programming is communication. In attempting to teach a programming language we are trying to provide the learner with a means of communication, a means of expressing himself. At first sight it will appear that the communication will be one way, between the program writer and the machine on which his program is processed. This view is too simplistic, for the communication occurs on a number of different levels.

Certainly it is important that a programmer is sufficiently familiar with the language he selects to write his program to produce concise and efficient code, but it should not be forgotten that, after successful development, a program may need to communicate with its user while executing. This aspect of communication is now, justifiably, receiving considerable attention. It is no longer satisfactory that a program produces the correct results – it should also be easy to use, and should be 'bulletproof', which is to say that, no matter how inaccurate the user's input, the program should always provide a sensible and intelligible response. In the jargon, the program should be 'user friendly'.

A further level of communication, all too often neglected, is that between program writer and program reader. Program writers frequently assume that the only readers of the program will be themselves and a computer. The consequence of this assumption is that the program may be tedious and difficult to assimilate by anyone given the task of modifying, or simply reading, the original. Like everything else of man's creation, software will not be perfect, and should be written with the knowledge that it will need to be maintained. This means taking all reasonable steps to ensure that the program logic is lucidly expressed by the text, and the layout and presentation of a program help considerably in this. Unfortunately, there are constraints imposed by some language implementations that inhibit good presentation. Thus when using a BASIC interpreter with access to a limited amount of memory, there will be pressure on a programmer to omit comments and to discard unnecessary spaces. We recognise the pressures, but regret their effect on the intelligibility of programs.

The concept of 'program style' encompasses the presentation, layout and readability of computer programs. The principles apply to any programming language, whether high level or low level. The factors that contribute to program style are undoubtedly highly subjective, and thus contentious. Our contribution to the debate is to enumerate what we consider to constitute a reasonable set of metrics, whose application can be automated, and to associate with each of the

program examples within the text a 'style score'. At the foot of every non-trivial program you will see this style score enclosed in square brackets. For small examples the style score can be sensitive to small changes in presentation, for example, the addition of a blank line. Nonetheless, we give it so that the reader can judge its usefulness. A small C program is illustrated in example I.1 to give a hint of what is to follow. The derivation of the style score is detailed in appendix 3. Suffice it to say here that the score is a percentage, and that the higher the score, the more 'elegant' the program.

The programming language C is a powerful language, and deserves its increasing popularity as one of the most important systems programming languages currently available. Without wishing to over-stress program style and the importance of good program design, we feel that it is necessary to point out that no programming language is, as yet, so powerful as to conceal flaws in program logic or to make its clear exposition unnecessary. Sound program logic is achieved by design, and in recent years considerable attention has been given to program design methods. The National Computing Centre has produced an excellent publication on the subject (Bleazard, 1976) which neatly summarises a wide variety of views. A further useful summary is the article by Weems (1978). Whether a structured program is achieved after the design stage will depend on the person or persons who translate the design into a program in an appropriate programming language – a not inconsiderable task. The book by Dahl *et al.* (1972), and the state of the art report (Bates, 1976) are both worthy of the reader's attention.

Programs can become such complex artefacts that many professionals in the computing field speak of software being 'engineered'. With this in mind, it is not surprising to find 'software tools' produced to assist in this engineering. The software tools philosophy espoused in Kernighan and Plauger (1976) and realised in UNIX* is an impressive demonstration of the importance of this approach. We believe that UNIX and C have significantly expanded our own computing horizons, and thoroughly recommend the experience to others.

Example I.1

```
main()

    /* to resort the letters of a word into alphabetical
       order - e.g. the basis of an anagram dictionary */

    { char word[20], min;
      int  i, j, pos, len;

      printf("Gimme a word ... ");
      scanf ("%s", word);
```

*UNIX is a Trademark of Bell Laboratories.

```
    len = strlen(word);
    for (i=0; i<len; i++)
        {
          min = '~'; pos = 0;
          for (j=0; j<len; j++)
              if (word[j]<min) { min=word[j]; pos=j; }

          printf("%c", min);
          word[pos] = '~';
        }

    printf("\n");
}
```

```
[ style 51.7 ]
```

There are a small but growing number of texts that describe UNIX and C. That by Bourne (1982) we have found particularly useful. Kernighan and Ritchie's (1978) book remains the definitive C reference, while the experienced C user might better himself by reading Feuer (1982).

In this book, the first chapter describes the structure of C programs. Chapter 2 introduces functions, contrasting them with macros. Chapter 3 deals with input and output, emphasising the importance of the interface between the program and its environment.

Chapters 4 and 5 explain the two features of any programming language that give it its power – the control constructs of conditional branching and looping. Operators are introduced in chapter 6, while chapter 7 illustrates the use of arrays and strings.

This is the point at which all the 'basic' features of C have been covered. The remaining chapters describe what we consider to be 'advanced' features – derived data types in chapter 8, data structures in chapter 9 and the C preprocessor in chapter 10. The final chapter presents some guidance on program 'style', which we could define loosely as that enigmatic quality that distinguishes adequate programs from superlative ones.

In learning any programming language we have found that examples which, as well as illustrating language features, stimulate the reader's interest are of particular importance. We have tried to present an interesting variety of examples. It is easy, however, to be left with the impression that all programs are small. To redress this imbalance we have presented a rather larger example which we refer to as RatC. RatC is a C program that accepts as input a program written in a subset of C and produces as output an intermediate code version of the program. In addition to making many references to RatC as a source of examples, we provide the user with sufficient information to implement his own small C compiler.

Above all, C is a language to enjoy. The kind of thing you always wanted to be able to do in other programming languages becomes possible in C – but be warned that its power, as well as getting you out of trouble quickly, can get you into trouble just as quickly.

We hope that learning C gives the same lift to your programming experience as it has done to ours.

1 Program Structure

In the introduction we attempted to show that programming must be undertaken in a disciplined and organised manner. If the resulting program is to display the benefit of this approach then the programmer must be thoroughly familiar with the program structure dictated by the programming language that he, or she, is to use.

FUNCTIONS

A C program consists of one or more functions. One of these functions must have the name *main*. A program is executed when the underlying Operating System causes control to be passed to the function *main* of the user's program. The function *main* differs from the other functions in a program in that it may not be called from within the program, and the parameters to *main*, if they exist, are provided by the Operating System. It is usual, but not essential, for *main* to be the first function of the program text.

Viewed simply, a function name is nothing more than a collective name for a group of declarations and statements enclosed in curly brackets or braces { }. The function *useless* below is of little value since it contains no executable statements. Its only purpose is to illustrate the appearance of a minimal function.

```
useless()
{
}
```

The parentheses following the function name are essential, and will later be shown to be more useful than the present example suggests.

If we assume that *main* is the first function defined in a C program text then, because no function may contain the definition of another function, the definitions of the subsidiary functions of the program text will follow. There may be only two or three such functions, in which case their purposes will be easy to determine, or there may be many, as in RatC. There is no special ordering of the functions dictated by the programming language C (in contrast to Pascal which, despite advocating the structured approach to problem solving, precludes its effective

use by insisting that all functions be defined before they are used). However, after emphasising the value of a program as a means of communication, it would be foolish to suggest that an arbitrary order for the functions would be as good as an order with some rationale. The function definitions could be arranged in alphabetical order, or they could be grouped according to their purpose. This latter ordering is not so easy to achieve but can frequently be more helpful. It is this ordering that we adopted for the functions that comprise RatC. Note, however, that even for a program of this size, if utilities are available to number the lines of the source program and provide a list of function names, together with the line numbers on which the definitions start, then it is easy to locate individual function definitions whatever their order of appearance.

IDENTIFIERS

An identifier in C, whether it represents a function name or a variable, consists of any sequence of the characters [a-z, A-Z, 0-9, _], of which usually only the first eight are significant. The first character of an identifier must not be a digit. Upper and lower case letters are distinct, so that, for example, the identifiers 'count', 'Count' and 'COUNT' represent three different quantities. Identifiers are characterised by the two attributes 'type' and 'storage class'. The type of an identifier determines the type of object that it will be used to represent; so, for example, *int*, *float* and *char* qualify an identifier as representing an integer, a real (or floating point) number and a single character respectively. The full list of available types is given in appendix 5. An identifier's storage class determines the way in which it can be accessed from other parts of the program.

FILES AND THE STORAGE CLASS *external*

A program of the size of RatC should prompt questions concerning whether it resides entirely in one file or whether the text is spread over several files. To illustrate the effect of file structure on C programs and the symbols or names used within them, consider the examples given below, in which items within the same file are enclosed by a box.

Example 1.1

file1.c

```
main()
{

}

function1()
{

}
```

file2.c

```
function2()
{

}

function3()
{

}
```

In example 1.1, if we ignore *main*, any of the three functions could legitimately contain references to each of the remaining two. *main* may call any of the other three functions. This is possible because all function names belong to the storage class *external*. Any symbol name from this storage class may be referenced across files.

A function may also contain a call of itself. This is known as a recursive call, and an example of such a call will be found in the number printing function *prnum* given as an example in appendix 1.

STORAGE CLASS *automatic*

In order that the functions we define can perform some useful role they will need to manipulate data. As in most programming languages the name and type of every data item must be declared. A declaration does not necessarily reserve storage to be associated with the identifier, but rather establishes the type and storage class of the declared identifier. In the example below 'size' is declared to be an integer and its storage class is *automatic*.

```
main()
{
    int size;
}
```

The identifier 'size' is local to the function *main* and may only be used within *main*. If the name 'size' is used in any other function in the program it is not then connected in any way with the data item of the same name in function *main*. The storage class is known as *automatic* because, for any identifier in the class, storage space is allocated when the function is entered and given up when exit is made from the function. This is the default storage class for any identifier declared within a function. While this form of storage is economical, in that it is needed only when a function is being executed, it does not meet all our requirements.

STORAGE CLASS *static*

Imagine that, as part of a check upon the operation of a program, it is necessary to count the number of times that a function was executed. The count should be local or private to the function but the associated storage should be preserved from one call of the function to the next in order that the count may be accumulated. An identifier with storage class *automatic* is clearly inappropriate, since its value would be lost between successive calls of the function. Consider example 1.2: the identifier 'count' has been defined as type integer with storage class *static*. It could be used to accumulate the number of calls of *function1*.

Example 1.2

file1.c

```
main()
{

}

function1()
{
    static int count;
}
```

file2.c

```
function2()
{

}

function3()
{

}
```

As another example, suppose that two or more functions are used to manipulate the contents of a table. Each function will require to access the table and its associated pointers. It might also be desirable to protect the table from corruption by ensuring that no other function of the program gains access to the table. Both requirements can be met by using data items belonging to the *static* storage class within the same file.

Example 1.3

file1.c

```
main()
{
    int size;
}

function1()
{
    int i;
}
```

file2.c

```
static int ptr;

function2()
{

}

function3()
{
    int i;
}
```

In example 1.3, the identifier 'size' can only be used in *main*. The identifier 'i' of *function1* has no logical connection with the identifier 'i' of *function3*. The second file contains the declaration of 'ptr'. Both *function2* and *function3* may use the identifier 'ptr', as may any other function defined in that file. The storage class of 'ptr' is not *automatic* but *static*. Identifier 'ptr' is not accessible to a function in any other file. Note that it is not only function names that belong to the storage class *external*. We can declare the names of other data items so that they belong to this class. These names too may be referenced across files. If we change file1 of our example by adding the line

```
extern int ptr;
```

and remove the word *static* from file2, as shown in example 1.4, then the function *main* can now reference the item 'ptr' defined in file2.

Example 1.4

file1.c

```
main()
{
     extern int ptr;
     int size;
}

function1()
{
     int i;
}
```

file2.c

```
int ptr;

function2()
{

}

function3()
{
     int i;
}
```

If, however, the *extern* statement were to appear as the first line in file 1 then all functions in that file could refer to 'ptr', and this would be the same object declared in file2. In distributing a program text across files in this fashion we would need to ensure that for each identifier name in the external storage class, other than function names, there was one declaration of this name that did not include the word *extern*. This is called the definition of the identifier. The prefix *static* must be omitted in this definition.

The discussion on files assumes that it is sensible and convenient to divide a program text in this manner and also that the names of the two or more files are passed to the C compiler for processing. There are circumstances, however, in which it might be convenient to divide our program physically between files but to treat it logically as a large program text in one file. This facility is made available by the C preprocessor.

THE C PREPROCESSOR

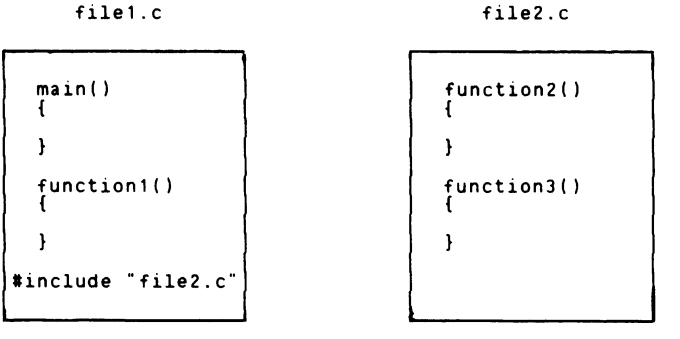
Preprocessing is, as its name suggests, undertaken prior to compilation and provides two important facilities; the ability to 'include' files and the ability to 'define' text for macro replacement. These are extremely convenient facilities and, since frequent use is made of them, they are introduced at this early stage.

#include

Example 1.5 differs from example 1.1 in the addition of one line at the end of file1. This is sufficient to change the organisation of the program in a small but significant way. The 'include file' request must appear at the left margin and is treated as a request to replace the line itself by the contents of the file given, in this case file2.c. Under the UNIX operating system, if the file name appears in double quote marks it is assumed to be in the current directory; if the file name is included instead in

angle brackets, a special directory is assumed to be the location of the file. In either case the contents of the file replace the *include* directive and the combined text is passed on to the C compiler which treats it logically as one file of program text. Several files may be coalesced by use of suitable *include* directives. Included files may themselves contain *include* directives. While this is a legitimate use of the included file facility, an included file more usually contains *define* directives. A file containing *define* directives is known as a header file and, by convention, has a filename ending in '.h'. Any file containing C program text has a name which ends with '.c'.

Example 1.5



#define

The *define* directive provides the user with a macro replacement facility. The C preprocessor in this context is a macro processor, although this is not always appreciated by newcomers to this facility. The most common use of the *define* directive is of the form

```
#define  DAYSINWEEK  7
```

The preprocessor will thereafter replace the text string 'DAYSINWEEK' throughout the entire text by the text string '7'. In one sense this facility can be likened to the *const* section of a Pascal program in that it provides a means of removing all explicit constants from a program text and enables the user to use symbolic names instead. We think that it is good practice to gather all such definitions at the head of the program text file. However the *define* directive is not restricted to use in the manner described above for program constants. It is, in general, much more powerful and useful, since it replaces one text string by another and will, as we shall see later, also deal with parameters.

SIMPLE C CONSTRUCTS

In order that we may use examples to illustrate the points made in the text, we need, as has already become obvious, some programming language constructs. Even the simple examples need to demonstrate that they work by printing something. We therefore introduce the *printf* function.

```
printf("The answer is 42");
```

printf, print formatted, is perhaps the most commonly used output function. Whatever text appears within the double quote marks is, with a few important exceptions, printed on the user's output device. Input and output statements are not an integral part of the C language, but are usually provided within a commonly accessible library of such routines, which will be made available to the program via an *include* file. Under UNIX, for example, use of

```
#include        <stdio.h>
```

at the head of a program is a convenient way of obtaining access to some commonly used definitions. These definitions include several of the simpler input/output functions. We shall assume for convenience that the user is using a visual display unit (VDU) to a multi-user or microcomputer system on which C is available.

```
printf("\nThe answer is 42\n");
```

This variant of the first *printf* statement prints a newline character, represented by the character pair \n, before and after printing the string itself. All statements in C are terminated by a semi-colon. There may be more than one statement per line. An assignment statement is exemplified by

```
answer=42;      /* 42 is a decimal constant */
answer=052;     /* leading 0 indicates an octal constant */
answer=0x2a;    /* leading 0x or 0X indicates a hex constant */
```

where we assume that 'answer' has been declared to be an integer. Lastly, let us note at this point that the braces { } may be used to enclose one or more C statements

```
{ question=99; answer=42; }
```

The collective name for statements enclosed in this way is a compound statement. It will become obvious from the examples that in C a comment is any text string enclosed by /* and */.

Further examples of the use of the *define* directive can now be given by using the *printf* function. The definition

```
#define  STARS  printf("**********")
```

will cause the symbol 'STARS' to be replaced by the call of the function *printf*. When viewed in the context of the example given below it will be appreciated that the *define* facility could save us some tedious typing.

```
#define  STARS  printf("**********")

main()
{
     STARS;
     printf("\nThe answer is 42\n");
     STARS;
}
```

'*defining*' VDU CHARACTERISTICS

We can use the *define* directive in another more useful way to improve the quality, and thus the user friendliness, of the output produced by any program. Most VDUs in common use have facilities to home the cursor, clear the screen, and so on. Invariably to use these features means sending a special character sequence to the terminal. The character sequence is not easy to remember unless one uses it constantly; it varies from one manufacturer's product to another and frequently between different models from the same manufacturer. What we suggest is that these codes are set up once and for all using *define* directives. For a Lear Siegler ADM5 we would have

```
#define  CLEAR  printf("\033Y")
#define  HOME   printf("\036")
```

Recall that the backslash followed by n was used to denote a newline character. Backslash followed by a number can be used in *printf* and elsewhere in a C program, to denote the character defined by the ASCII code in octal which follows the backslash. A table of the ASCII characters with their octal representations is given in appendix 5. To clear the screen of this particular terminal we can send the escape character (ESC) followed by the letter Y. Since this clears from the cursor to the end of the screen, the HOME command should precede the CLEAR. This form of CLEAR command is given because ESC followed by a character sequence is a common way of expressing VDU directives.

The number of special features available on a VDU varies considerably. A VT100 terminal, for example, will offer cursor addressing, blinking, highlighting, reverse video and other features all of which are selected by a special character sequence beginning ESC[. For any VDU these special features should be noted and appropriate *define* directives set up as illustrated in the examples. Thereafter all the *define* directives for one terminal should be collected together in a suitably named file. Any C program wishing to use these facilities need then only *include*

this file at the head of the program and all the commands defined for that VDU become available. The contents of two such *include* files are given in appendix 4.

SUMMARY

In this chapter we have described the structure of C programs. We have illustrated the convenient and versatile mechanisms that are easily available to the programmer to help produce a well-organised and a well-structured program. We shall endeavour to reinforce these ideas through the examples that we present. Our presentation may not be perfect and may seem for the smaller examples to dominate the examples themselves. Effort spent on organisation, structure and layout of a program is worth while and we hope that this point is adequately demonstrated by RatC. Considerable effort has gone into the organisation and presentation of this the largest example in the book. If you find it easy to assimilate and find your way round, then use some of the same strategy on your programs. If on the other hand you feel the presentation or organisation could be improved, then learn from our failings and produce well-structured programs as a result.

2 Functions

As we have seen in the previous chapter, functions offer an easy way to construct a modular program. Since they are such an essential part of good C programming we shall introduce their facilities at an early stage to encourage familiarity with their use.

In order that our examples may achieve something, even if it is not especially useful, we will make use of the *printf* statement introduced earlier.

Example 2.1

```
#include "adm5.h"
#define  GAP printf("\n\n\n\n")

      /* a program to print large letters */

main()
    {
      HOME; CLEAR; GAP;       /* clear the screen */
      bigH();       GAP;
      bigI();       GAP;
    }

/* bigH prints H as a 7*5 matrix of asterisks */

bigH()
    {
      printf("*   *\n");
      printf("*   *\n");
      printf("*   *\n");
      printf("*****\n");
      printf("*   *\n");
      printf("*   *\n");
      printf("*   *\n");
    }

/* bigI prints I as a 7*5 matrix of asterisks */
```

```
bigI()
    {
      printf("*****\n");
      printf("  *  \n");
      printf("  *  \n");
      printf("  *  \n");
      printf("  *  \n");
      printf("  *  \n");
      printf("*****\n");
    }

[ style 55.6 ]
```

Because the program does not do much, its structure, and the preprocessor facilities that it uses, are easily seen. The *include* file 'adm5.h' contains screen control instructions for a Lear Siegler ADM5.

In the body of the program, after clearing the screen, a call to the function *bigH* is made. When executed this function causes asterisks to be printed representing the character H in a 7 * 5 matrix of characters. Similarly *bigI* causes the character I to be printed. The symbol 'GAP' ensures an appropriate separation between the characters and whatever follows them on the screen.

Anyone choosing to type example 2.1 into their own machine will quickly realise that they are typing identical *printf* statements several times over. Repetition like this should always prompt the question, 'Is there a better way?' The answer is often 'yes', and frequently there is more than one 'better way'. Example 2.2 illustrates that by using the *define* facility of the preprocessor we can save writing and typing of text. Remember that the preprocessor will simply replace the defined symbol by its definition throughout the program text, and so the version of program 2.2 that reaches the compiler will be logically equivalent to program 2.1.

Example 2.2

```
#include "adm5.h"
#define  GAP printf("\n\n\n\n")

     /* allstars prints all stars */
#define allstars printf("*****\n")

     /* endstars prints end stars */
#define endstars printf("*   *\n")
```

```
     /* midstar prints mid star  */
#define midstar printf("  *  \n")

main()
    {
      HOME; CLEAR; GAP;       /* clear the screen */
      bigH();       GAP;
      bigI();       GAP;
    }

bigH()
    {
      endstars; endstars; endstars;
      allstars;
      endstars; endstars; endstars;
    }

bigI()
    {
      allstars;
      midstar; midstar; midstar; midstar; midstar;
      allstars;
    }


[ style 63.1 ]
```

Alternatively, the program can be rewritten using function calls instead of *defines* by declaring *allstars*, *endstars* and *midstar* as functions, as shown in example 2.3. The programs 2.2 and 2.3 are functionally, but not logically, equivalent, in the sense that, although the output from both is the same, in one case it is produced by a program with three functions, and in the other, by a program with six.

Example 2.3

```
#include "adm5.h"
#define  GAP printf("\n\n\n\n")

main()
    {
      HOME; CLEAR; GAP;       /* clear the screen */
      bigH();       GAP;
      bigI();       GAP;
    }
```

```
bigH()
    {
      endstars(); endstars(); endstars();
      allstars();
      endstars(); endstars(); endstars();
    }

bigI()
    {
      allstars();
      midstar(); midstar(); midstar(); midstar(); midstar();
      allstars();
    }

        /* allstars(), endstars(), midstar() */
        /* are now defined as functions      */

allstars()
    { printf("*****\n"); }

endstars()
    { printf("*   *\n"); }

midstar()
    { printf("  *  \n"); }

[ style 47.3 ]
```

MACROS OR FUNCTIONS?

When executing, the program 2.3 produces the same results as the two previous versions of this program. Which is best depends on what criteria are used for the comparison. In example 2.2 the preprocessor replaces all symbols defined in a *define*. The transformed program is passed to the C compiler. When executed, the body of the function *bigH* causes seven *printf* statements to be obeyed. When executing the function *bigH* of 2.3, seven function calls are executed and each call causes a *printf* statement to be obeyed. For examples of this size we are unlikely to notice the difference in compile time or execute time between 2.2 and 2.3. If we were able to measure such times accurately then we would find that 2.2 compiled more slowly than 2.3, but executed more quickly. Our guideline, while approximate, will be that where symbols are replaced by small amounts of text then the symbol will be defined in a *define* statement, otherwise the symbol will be defined as a function. In contrast, if we knew that a function with a small

body was called in a part of the program that was heavily used, then we would consider replacing the function definition by a *define* statement for the symbol name. This would save the overhead of the function call at execution time. Decisions like these are reflected in the definition of some of the symbol names used in the RatC compiler.

USING PARAMETERS

Functions are much more useful if we are able to pass information to them. Information can be passed implicitly, by using within the function symbol names that are defined elsewhere, or explicitly, by using parameters. The examples of *printf* used to date have been limited in that they simply print a given string. However, *printf* is a much more versatile function than these early examples suggest. In particular it can be made to print the value of data items that are passed as parameters, thus

```
printf("%c   %c\n" , '*','*');
```

The first parameter must always be the string (in double quotes) that contains characters to be printed, formatting information, and conversion characters. The percent sign % precedes conversion characters in the string. More details of the conversion characters will be given in chapter 3. For the moment it will be enough to know that the letter c after % indicates a character conversion. For each conversion character in the control string a suitable parameter must be provided within *printf* following the control string. Each parameter following the control string must have a corresponding conversion character within the control string. The *printf* statement given above has exactly the same effect as the *printf* statement given in function *endstars* of 2.3. We are now in a position to add a useful parameter to those functions that we have defined.

DEFINING PARAMETERS

Consider the following version of *endstars*

```
endstars(anychar)
char  anychar;
{
    printf("%c   %c\n", anychar, anychar);
}
```

Here the function, *endstars*, is defined as having a parameter. A parameter, such as 'anychar', which is used in the function definition is called a formal parameter. The type of the parameters, if there is one or more, is defined before the brace

which marks the start of the function body. The parameter may then be used in a manner consistent with its definition anywhere within the function body. The function *endstars* simply uses 'anychar' as a parameter to *printf*. Hence whatever character is passed to *endstars* through the parameter list in a function call is printed in the manner that should now be familiar.

Example 2.4

```
#include "adm5.h"
#define  GAP printf("\n\n\n\n")

main()
    {
      HOME; CLEAR; GAP;       /* clear screen */
      bigH('H');   GAP;       /* use H to construct letter H */
      bigI('I');   GAP;       /* use I to construct letter I */
    }

bigH(ch)
    char ch;
    {
      endstars(ch); endstars(ch); endstars(ch);
      allstars(ch);
      endstars(ch); endstars(ch); endstars(ch);
    }

bigI(ch)
    char ch;
    {
      allstars(ch);
      midstar(ch); midstar(ch);
      midstar(ch); midstar(ch); midstar(ch);
      allstars(ch);
    }

        /* allstars(), endstars(), midstar()   */
        /* are now defined as functions,       */
        /* each has one parameter of type char */

allstars(ch)
    char ch;
    { printf("%c%c%c%c%c\n", ch, ch, ch, ch, ch); }
```

```
endstars(ch)
    char ch;
    { printf("%c   %c\n", ch, ch); }

midstar(ch)
    char ch;
    { printf("  %c  \n", ch); }

[ style 61.1 ]
```

If all the functions of the example 2.3 are parameterised in this fashion, and the corresponding calls are suitably amended, then we obtain a program such as 2.4. This program is more versatile than the others in the series in that by changing the character that is the actual parameter to *bigH*, or to *bigI*, we can change the output produced. Using parameters in this way will usually help to make quite clear what must be passed from the caller to the function. If communication between a caller and a function is done implicitly by use of symbols to which both have access, the communication is not so obvious to the reader. For this reason early examples within the book will use the parameter list. Later examples will not be restricted in this way.

A further example of a function with parameters is one that enables us to move the cursor on the VDU screen to any position. For the ADM5 this function definition might appear as

```
    /* to move the cursor to `row', `pos' */

cursor(row, pos)
int  row, pos ;
{
    int us = 31;    /* initialise for ADM5 */
    printf("\033=%c%c", us+row, us+pos);
}
```

The call

```
cursor(1, 1);
```

would move the cursor to the 'home' position, while the call

```
cursor(12, 40);
```

would move the cursor to the middle of the screen. However, all of our other screen control directives are gathered together in an *include* file. The logical place for *cursor* is within that file too. But *cursor* needs parameters and so far none of the symbols in a *define* directive has used parameters. Recall that replacement of defined symbols is undertaken by a macroprocessor and, fortunately, this offers us parameter replacement. Hence the addition to our file of the following definition

```
#define  CURSOR(r, p) printf("\033=%c%c", 31+r, 31+p)
```

will perform exactly the same role as the function of the same name. We will therefore extend both screen control files to contain the same cursor movement feature, but note that it is implemented in quite a different way for the VT100 (see appendix 4).

USING *return*

As well as passing information to a function, we must be able to pass information back to the caller from the function. This may be done in one of three ways: by using a *return* statement to pass a value via the function name, by passing one or more values back through the parameter list, or by changing the values of symbols to which both the function and the caller have access. For the reason given earlier this last form of communication will not yet be used.

The function *surface* in example 2.5 computes the surface area of a rectangular box having dimensions that can be expressed as integers. The value computed is communicated to the caller by the *return* statement and can be thought of as being associated with the function name. The function call can, in consequence, be used in expressions. In particular the call may appear in a *printf* statement, as indicated in example 2.5.

Example 2.5

```
main()
    { int length, width, depth;

      length = 10 ; width = 16 ; depth = 4;
      printf("surface area = ");
      printf("%d\n", surface(length, width, depth));
    }

    /*******************************/
    /* to compute the surface area */
    /*     of a rectangular box    */
    /*******************************/

surface(len, wid, dep)
    int len, wid, dep;
    {
      return(2*(len*wid + wid*dep + dep*len));
    }

[ style 53.8 ]
```

Even such an apparently simple example raises several new points. The conversion character following the percent sign is d to indicate a decimal integer. In other respects the *printf* statement is little different from those already seen. The function definition has three formal parameters of integer type (*int*). The function call has three actual parameters of integer type. The formal parameters and actual parameters correspond in order, number and type. The function body consists simply of a *return* statement which computes the surface area. So that no confusion arises in these early examples, the formal parameters have been given names that are different from the names of the actual parameters. The names leave no doubt as to which formal parameter corresponds to which actual parameter.

The *return* statement passes a single value from the function to the caller. The type of this value is determined by the form of the expression in the *return* statement and the type of the operands. If the returned value is of type integer or character (*char*) then the function definition is as given in example 2.5. However if the parameters to the function were of type *float* then the program should appear as in example 2.6.

Example 2.6

```
main()
    { float length, width, depth;

      float surface();      /* NB  this is needed */

      length = 20.0; width = 26.0; depth = 4.0;
      printf("surface area = ");
      printf("%f\n", surface(length,width,depth));
    }

        /* this version of surface returns */
        /* a result of type float          */

float surface(len, wid, dep)
    float len, wid, dep;
    { return(2*(len*wid + wid*dep + dep*len)); }


[ style 51.9 ]
```

The type of the actual parameters and the formal parameters has been changed to *float*. The function must now return a value that is also of type *float*. The type of result returned by the function is signalled by preceding the function name in the function definition by the type of value to be returned. There is a further consequence of this action. A function is assumed by default to have the type *int*. If it is our intention to use a function that violates this assumption then we must

signal this intention. This is done by including, in the functions or files that call this function, a declaration of the function. It is for this reason that an additional line appears

```
float surface() ;
```

Another example of a function that only has a *return* statement for a body is the function *numeric* in RatC. This function returns a character value.

RETURNING VALUES VIA THE PARAMETER LIST

As well as receiving data values through the parameter list it is also reasonable to expect that we can communicate data values back to the caller through one or more parameters. In order to understand the mechanism by which this is achieved, let us observe that in C all parameters are value parameters. That is, the values of the actual parameters are copied into temporary storage in the function work space upon entering the function. Thereafter, the function only makes reference to these local values. If assignment is made within the function body to one of the parameters, it will be the local copy that is changed, not the original. At first sight this seems to inhibit communication from the function to the caller via the parameter list. For C the way out is to use the address of the relevant data item.

ADDRESSES AND POINTERS

Address	Contents
&i	i
ptr	*ptr

In a high-level language it is not usually necessary to know or care about the address in memory of the data values that we wish to manipulate. As a consequence, in some languages we have to resort to subterfuge in order to access specific memory locations. Pascal is one such language. At the other extreme, if it is too easy to access and modify memory locations then a program exploiting this facility can become unreadable. Thus a BASIC program which makes too much use of 'peek' and 'poke' instructions is not easily intelligible. In C an easy and convenient way of obtaining the address of a data item is provided. Correspondingly, given the address of a data item, we can easily obtain its value. As might be expected in C the mechanism is short and simple. We obtain the address of an item by prefixing it with ampersand: thus &x is the address of x. In order that we can manipulate addresses we need to be able to define items that have pointers or addresses as their values. This is done as follows

```
int i ;      /* i represents a value of type integer */
int *ptr;    /* ptr holds the address of a data item */
             /*      of type integer.                */

             /* an alternative declaration with the  */
             /* same effect follows                  */
int i, *ptr;
```

This notation can now be used to enable a function to communicate with its caller. For if the caller passes to the function the address of a data item, it is the address that is stored in the local storage area of the function. The function cannot change the address of the item, but it can change the contents of the address which is, after all, what we wish to happen. Example 2.5 may now be rewritten as example 2.7.

Example 2.7

```
main()
    { int length, width, depth, area;

      length = 10; width = 16; depth = 4;
      surface(length, width, depth, &area);
      printf("surface area = %f\n", area);
    }

      /* the fourth parameter is an address */
      /* we refer to its contents as *addr  */

surface(len, wid, dep, addr)
    int len, wid, dep, *addr ;

    { *addr = 2 * (len*wid + wid*dep + dep*len); }


[ style 45.8 ]
```

The differences between this example and the two previous examples need to be highlighted. In the function *surface* the formal parameter 'addr' is used to communicate the computed surface area back to the caller. In order that this may happen the content of 'addr', *addr, is typed as an integer which means that 'addr' is an address. The caller must therefore provide the address of an integer type variable as the fourth parameter. In the example it is the address of 'area', &area, that is provided. Since a *return* statement is not used within the function no value is associated with the function name. Accordingly the function call is a statement in the main segment of the example.

When a function has only one value to communicate to the caller it will usually be convenient to use a ***return*** statement to pass the value via the function name. If more than one value is to be communicated to the caller, then we can use both the return mechanism and the parameter list, or we can use the parameter list alone. Functions exhibiting these features will be used later in the book when further language constructs have been introduced.

SUMMARY

In this chapter, we have introduced two methods of abbreviating the number of statements that a programmer must write to produce a program: *defines* and functions. Choosing between the two is largely a matter of personal taste, subject to the guidelines that we have laid down.

Functions represent a major aid both to the modular development of a program and to its subsequent readability. The length of a function is again a matter of taste; ideally, a function should perform a single task, and should rarely, if ever, exceed a printed page in size.

We have discussed the various methods by which the functions of a program can communicate with each other. Suitable use of parameters not only generalises the use of a function, but also assists in an understanding of its purpose and the extent to which different parts of a program fit together.

The RatC compiler is a good example of the judicious use of each of these features.

3 Output and Input

OUTPUT

Our use of the output function *printf* has so far been straightforward. We have seen that, as well as printing text strings, it can easily be made to convert the internal form of our data items into a suitable form for printing. The general form of the *printf* function call can be expressed as

```
printf(control_string [, argument_list])
```

(The square brackets enclose an item that is optional.) The control string may contain characters to be printed, control characters preceded by backslash, and conversion specifiers.

CONVERSION SPECIFIERS

For each conversion specifier there must be a corresponding argument in the argument list. The minimal form of a conversion specifier is a percent sign followed by one of a limited set of characters. Examples of conversion specifiers are given in table 3.1.

The general form of the conversion specifier can be written

```
%[-][fw][.][pp]C
```

where C is the conversion character or character pair.

- is used to indicate that the output is to be left justified in the field (the output is right justified by default).

fw is a digit string giving the minimum field width – the total number of print positions occupied. Excess places in the field are by default filled with blanks. If the first digit of the field width is zero, the field is zero filled. A data value that is too large for the field specified is printed in its entirety.
(An asterisk used instead of the digit string signifies that the field width

is given by an integer (constant or variable) in the appropriate position in the argument list.)

separates fw from pp.

pp is a digit string which for a data item of type *float* or *double* specifies the number of digits to be printed after the decimal point. For a string it specifies the number of characters from the string to be printed.

Table 3.1

Conversion characters	Argument type	Comment
c	char	Single character
d	int	Signed (if −ve) decimal
ld or D	long	Signed (if −ve) decimal
u	int	Unsigned decimal
lu or U	long	Unsigned decimal
o	int	Unsigned octal, zs
lo or O	long	Unsigned octal, zs
x	int	Unsigned hexadecimal, zs
lx or X	long	Unsigned hexadecimal, zs (zs . . . zero suppressed)
f	float or double	Decimal notation
e	float or double	Scientific notation
g	float or double	Shortest of %e, %f
s	string	

Any invalid conversion character is printed!

The examples in the text so far have used none of the option facilities listed above. If our programs are to produce acceptable output then we must be able to take full advantage of the facilities offered by *printf*. Much the best way to obtain the necessary familiarity is to use, and experiment with, different conversion specifiers. To help in this a list of examples is given in table 3.2.

BACKSLASH

Within the control string we have used the backslash character preceding n to force the printing of a newline. There are other characters which have special significance when preceded by the backslash. The full list is given in table 3.3.

Table 3.2

Value	Control	: Output :
360	% 10d	: 360:
−1	% 10ld	: -1:
360	% −10d	:360 :
−1	% 10u	: 65535:
−1	% 10lu	:4294967295:
360	% 10o	: 550:
−1	% 10lo	:37777777777:
360	% 010o	:0000000550:
360	% −10x	:168 :
−1	% −10lx	:ffffffff :
360	% −010x	:1680000000:
3.14159265	% 10f	: 3.141593:
3.14159265	% 10.3f	: 3.142:
3.14159265	% −10.3f	:3.142 :
3.14159265	% 10.0f	: 3:
3.14159265	% 10g	: 3.14159:
3.14159265	% 10e	:3.141593e+00:
3.14159265	% 10.2e	: 3.14e+00:
programmer	% 10s	:programmer:
programmers	% 10s	:programmers:
programmer	% 10.7s	: program:
programmer	% −10.7s	:program :
programmer	% 10.4s	: prog:
programmer	% 10.0s	:programmer:
programmer	% .3s	:pro:

Table 3.3

\b	backspace
\f	form feed
\n	newline (line feed)
\r	carriage return
\t	tab
\ddd	ascii character code in octal
\'	'
\\	\

The features of the *printf* statement that have been itemised are sufficient to provide the user with good control over the output generated. Remembering also that through the control string itself we can separate one field from another, we appear to have everything that we need. It is now easy to modify example 2.1 so that it will print its large letters in the middle of the screen instead of on the left-hand side. All that is necessary is to ensure that, say, thirty-six leading spaces are printed before every string that is printed. This could be done by changing the first %c of each control string to %37c. If this proved unsatisfactory for some reason we would need to change each occurrence of 37 to something new. It will be much more convenient to use a *define* directive of the form

```
#define indent printf("%36c", ' ')
```

which will give us 36 leading spaces, and place the statement

```
indent;
```

before each of the relevant *printf* calls. A change in the number of leading spaces is now conveniently obtained by changing the value of one numeric constant.

INPUT

So far our primary concern has been the organisation of our output. We must also be able to supply our program with data when it is executing. Corresponding to the output function *printf* is the input function *scanf* which has a similar philosophy. If we continue with the assumption that input and output are done through a VDU then a call to *scanf* of the form

```
scanf("%d %d %d", &length, &width, &depth);
```

could have been used in example 2.5 to give values to the identifiers. The user would then need to type three integers as input when the program started to execute. Notice that because *scanf* must be able to communicate the input values to the caller, the caller must provide the address of the symbols to which the values are to be assigned. The general form of *scanf* is

```
scanf(control_string [, argument_list])
```

Within the control string blanks, tabs or newlines (collectively known as 'white space') are ignored. If any characters, apart from those needed in the conversion specifiers, appear in the control string, it is assumed that they are to match the next non-white-space character of the input stream. In particular, if any such characters appear as the first items in the control string then *scanf*, whenever it is called, will expect to find just these characters as the next to be read from the input stream.

CONVERSION SPECIFIERS

For *scanf* the conversion specifier has the following general form

```
%[*][dd]C
```

where C is the conversion character, * is an optional assignment suppression character, and dd represents a digit string giving the maximum field width. The character or character pairs admissible as conversion characters are given in table 3.4.

Table 3.4

Conversion characters	Argument type
c	Pointer to char
h	Pointer to short
d	Pointer to int
ld or D	Pointer to long
o	Pointer to int
lo or O	Pointer to long
x	Pointer to int
lx or X	Pointer to long
f	Pointer to float
lf or F	Pointer to double
e	Pointer to float
le or E	Pointer to double
s	Pointer to array of char
[.....]	Pointer to array of char

Consider the following simple example

```
scanf ("%d",&fw);              /* read an integer `fw'  */
printf("%*c\n", fw, '+');      /* print a plus sign in  */
                               /* a field width of `fw' */
```

An input field is normally delimited by white space characters, and hence for our first example of the use of *scanf* the three integers required for input could have been typed on a line separated by one or more spaces, or they could have been

typed one per line. Either form, or a mixture of the two, would be acceptable. Be warned that this means that *scanf* will read across input lines to find the next item of data. If the conversion specifier includes the assignment suppression character, no assignment is made; in other words the corresponding input field is matched and skipped. Should the length of the input field exceed the fieldwidth specified, then the data item is assumed to consist of the first 'fieldwidth' characters. Example 3.1 will perhaps help to clarify some of these points.

Example 3.1

```
char  ch;
char  string[20];
int   i, j, number, extension;
float x;

    /* assume the input string          PHONE65201X4133          */

scanf("PHONE %ld %c %d", &number, &ch, &extension);
    /* yields number = 65201, ch = 'X', extension = 4133        */

scanf("PHON %c %f %*c %d", &ch, &x, &ch, &extension);
    /* yields ch = 'E', x = 65201.0, extension = 4133           */

scanf("PHONE %2d %3d %c %2f", &i, &j, &ch, &x);
    /* yields i = 65, j = 201, ch = 'X', x = 41.0               */

scanf("%[^X] %c %d", string, &ch, &extension);
    /* yields string = "PHONE65201", ch = 'X', extension = 4133 */

scanf("PHONE %[0123456789] %c %d", string, &ch, &extension);
    /* yields string = "65201", ch = 'X', extension = 4133      */
```

Note that in the third example *scanf* has not read the last two characters (33) of the input stream. The next call to *scanf* would scan from the first of these characters. If the input stream contains nothing to match the current item of the control string, *scanf* terminates. Termination also occurs when all elements of the control string have been satisfied.

A variation on the string conversion specification is introduced in the last two examples, where the string is not delimited by white space characters. The specifier % [. . .] indicates a string containing any of the characters within the square brackets (and delimited by any that is not), while the specifier %[^ . . .] indicates a string delimited by the character set within brackets.

scanf returns to the caller the number of data items that were matched and assigned. A value of zero is returned when the next character of the input stream does not match the first item in the control string, and the value EOF (conven-

tionally defined in stdio.h) is returned when end of file is encountered. Thus if the call to *scanf* in the third example appeared instead as

```
items = scanf("PHONE %2d %3d %c %2f", &i, &j, &ch, &x);
```

then 'items' would be assigned the value 4.

The input stream searched by *scanf* is the standard input stream 'stdin'. The output produced by *printf* is directed to the standard output 'stdout'. It will frequently be necessary to scan other data sources and to direct output to other destinations. This can easily be achieved by using variants of *scanf* and *printf*. One of these variants allows us to deal with strings.

STRINGS

In C a string constant is a sequence of characters enclosed in double quotes. Like other data items strings may be read in, stored, manipulated and printed. Strings are stored in arrays of characters (this topic is covered in detail in chapter 7) and are referenced by the address of the first character, a pointer to array of char. The general form of the version of *scanf* that processes strings is

```
sscanf(data_string, control_string [, argument_list])
```

sscanf scans the string data_string attempting to match the data items specified in the control string. Successful matches are, when appropriate, assigned to the arguments in the argument list. Correspondingly

```
sprintf(data_string, control_string [, argument_list])
```

writes the arguments specified in the argument list into the data string in the manner determined by the control string. Since we can refer to strings only by means of a pointer to an array of char, it is obvious that the first argument to *sprintf* is the address of the data item that is to be changed.

I/O FUNCTION LIBRARY

The definition of the language C does not include the definition of input and output facilities. Instead, it is assumed that in every environment in which C programs are processed there will exist a library of functions to perform various input/output tasks. We assume that *printf* and *scanf* will be in this library. In passing we note that RatC uses neither *printf* nor *scanf*. The only functions that it uses are those that will read or print characters. The rationale for this is easy to appreciate as RatC is meant to be able to compile itself. In order to do this the RatC compiler must be able to process correctly all function calls that appear in the text being processed.

Functions that are used but not defined are assumed to belong to the runtime library available on the host machine. It will be relatively easy to provide, as part of this library, functions that read a character or print a character, whereas *printf* and *scanf* will necessarily be much more complex. Hence RatC assumes that they are not in the runtime library. The functions *getchar* and *putchar* should be part of any C library and, as their names imply, they communicate single characters from and to the VDU which we are assuming to be our input/output device. For example

```
ch = getchar();          /* get next character */
putchar(ch);             /* print it           */
```

or, equivalently

```
putchar(ch = getchar());
```

since, in C, an assignment is an expression that yields the value assigned as its result.

The input/output functions that will usually form part of any runtime library are listed in table 3.5. Any function not appearing and thought to be important or useful can be added to such a library by the user. There is no suggestion that the list gives all, and only, those functions that should appear in the library. When viewed collectively the functions listed in table 3.5 leave one wondering why

(1) the names *putc*, *getc* are not *fputc*, *fgetc* to indicate that they communicate with files, and

(2) the file_pointer argument of *putc*, *fputs*, *fgets* does not appear as the first argument as it does in *fprintf*, *fscanf*.

The following definitions might help the user whose sense of order is offended.

```
#define fputc(f, a)            putc(a, f)
#define fgetc(f)               getc(f)
#define fputstring(f, a)       fputs(a, f)
#define fgetstring(f, a1, a2) fgets(a1, a2, f)
```

FILE I/O

We have explicitly assumed so far that our input or output takes place from or to the user's terminal. While this will suffice for much initial work, we will wish, ultimately, to be able to read from and write to files. There are three files that are always available to any program. These are 'stdin', 'stdout', and 'stderr', the files for standard input, standard output and standard error messages. In practice these three files are always linked to the user's terminal. These files are opened at pro-

Table 3.5 Functions commonly appearing in the I/O library

printf(control_string [, argument_list])
scanf(control_string [, argument_list])
putchar(argument)
getchar( )
sprintf(data_string, control_string [, argument_list])
sscanf(data_string, control_string [, argument_list])
fprintf(file_pointer, control_string [, argument_list])
fscanf(file_pointer, control_string [, argument_list])
putc(argument, file_pointer)
getc(file_pointer)
fputs(argument, file_pointer)
fgets(argument1, argument2, file_pointer)

gram entry and closed at program exit. A user wishing to use other files must perform the opening and closing himself. Functions are provided to simplify this work. Opening a file involves passing a file name together with other information to the function *fopen* which returns a pointer to a file. Input/output functions using this pointer may write to a file or read from a file. The functions *fprintf* and *fscanf* are, apart from the fact that they communicate with a file, identical in action to their counterparts *printf* and *scanf*. The general form of their calls is given in table 3.5.

CLOSING A FILE

As part of the housekeeping associated with our program, a file should be closed when it is no longer needed. This is done by a call to the function *fclose* which has a general form

```
fclose(file_pointer)
```

When a program terminates normally, all open files are closed automatically.

OPENING A FILE

The operating system under which a C program executes may impose a limit on the number of files that the program may have open at one time. You should establish whether such a limit exists for your system and ascertain its value. If this limit is

inadvertently exceeded, a warning should be given when opening the file that causes the limit to be passed. Since other problems also could arise in opening a file, such as 'file does not exist', 'file is write protected', and so on, it is worth having a closer look at the details of opening a file.

A file pointer points to a data item that we have not so far encountered, an object of type FILE. This is not a simple data item such as one with type *char* or *int* that we have used previously, but is more complex. We need not know what data items the type FILE embraces. On UNIX systems, and other systems too, a file of standard definitions of items essential to the input/output functions is kept in the include file 'stdio.h'. By including this file in our program, we define such symbols as FILE, EOF and NULL. For local use within the program we need a file pointer, which we will call 'fptr', and we need to use *fopen* to open the required file. The general form of a call to *fopen* is

```
fopen(file_name, file_mode)
```

This function returns a file pointer to the file that has been opened. Since the function is therefore not returning a value of the default type (*int* or *char*), it must be declared within the function, or file, that is to use it. This is the reason for the line

```
FILE *fptr, *fopen();
```

in our modified program of example 3.2.

Example 3.2

```
#include "adm5.h"
#include <stdio.h>
#define GAP fprintf(fptr, "\n\n\n\n")

#define allstars fprintf(fptr, "*****\n")
#define endstars fprintf(fptr, "*   *\n")
#define midstar fprintf(fptr, "  *  \n")

FILE *fptr, *fopen();

main()
    {
      fptr = fopen("results.text", "w");
      if ( fptr == NULL )
          {
            printf(" error in opening file\n");
          }
```

```
        else
            {
              HOME; CLEAR; GAP;
              bigH();      GAP;
              bigI();      GAP;
              fclose(fptr);
            }
    }

bigH()
    {
      endstars; endstars; endstars;
      allstars;
      endstars; endstars; endstars;
    }

bigI()
    {
      allstars;
      midstar; midstar; midstar; midstar; midstar;
      allstars;
    }

[ style 67.5 ]
```

The file_name argument to *fopen* must be a string giving the name of the file to be opened. The file_mode argument must also be a string which specifies the type of access required. Possible file modes are

"r" read access

"w" write access

"a" append access

An attempt to open a file that does not exist for writing or appending will result in the file being created. If a non-existent file is opened for reading, then *fopen* will return the value NULL. Other errors will also result in the NULL value being returned by *fopen*. As a result, if the file is opened by a statement such as

```
fptr = fopen("results.text", "w");
```

we must immediately check that the file pointer 'fptr' is not NULL. This is done using a conditional statement, and while this has not yet formally been introduced, it should be clear from the example that a NULL return from *fopen* will cause our program to print an error message. A non-NULL return from *fopen* will cause our program to continue execution normally.

There are some specific comments worth making about the example 3.2. HOME and CLEAR have not been modified and so send their character sequences to the VDU and not to the results file. The FILE declaration must not be within a function since *main*, *bigH*, and *bigI* all need to refer to 'fptr'. *printf* has been changed to *fprintf* in the *define* directives and 'fptr' has been added as the first parameter. The standard input/output definitions in 'stdio.h' have been included.

SUMMARY

Output and input provide the interface between the program and its environment. A number of contemporary languages recognise that the environment is usually so implementation dependent that it is difficult to include these facilities within the language definition itself, and opt instead to provide them as library routines. The input/output facilities that we have discussed in this chapter are generally accepted as a *de facto* standard, but your local implementation should be checked before assuming that you can use the functions we have specified: your implementation may have either more or less than ours.

Since the principal function of all programs is to communicate, whether it be with other programs, devices, or the human user, as much thought should be given to the design of this interface as to the problem solution. It is not sufficient that a program produces the correct results, if those results, by virtue of poor presentation, are difficult to interpret; nor is it sufficient that a program assumes the integrity of its input, for this is usually the one factor over which the programmer has no control.

4 Decisions

A programming language that only offered the possibility of moving from one instruction to the next instruction in sequence would be extremely limiting. To be useful, we must be provided with the facility to choose different courses of action under different circumstances. There are two distinct ways that this may be done in C. We can use either the conditional statement or the *switch* statement.

CONDITIONAL STATEMENT

Two forms of the conditional statement are available in C

```
if (expression) statement1
if (expression) statement1 else statement2
```

An example of the latter form appears in example 3.2 to test that a file has been opened satisfactorily.

If the conditional statement currently under discussion is included, the kind of statements used so far in the text include

an assignment statement,
a function call,
a conditional statement,
a return statement, and
a compound statement.

(Recall that a compound statement is a group of statements enclosed by braces { }). Any of the statement types listed can be used as indicated by the general form of the conditional statement. Other forms of statement, defined later, can also be used. With the exception of the compound statement in the list above, all statements are terminated by a semi-colon. Anyone familiar with Pascal will find that the form of the conditional statement which uses *else* can, in certain circumstances, look strange. Different forms of the conditional statement are shown in example 4.1.

Example 4.1

```
if ( n<0 )  printf("n is negative\n");
if ( n==0 ) printf("n is zero\n");
if ( n>0 )  printf("n is positive\n");

/*  since  the  three statements above are  */
/*  distinct  conditional  statements, all  */
/*  tests are always performed. In contrast */
/*  consider the following alternative:     */

if ( n<0 ) printf("n is negative\n");
else if ( n==0 ) printf("n is zero\n");
else printf("n is positive\n");
```

What follows the comments in example 4.1 is a single conditional statement. The first *if* has a corresponding *else*, and what follows the *else* is a conditional statement. This way of expressing a condition may at first seem strange, but it will usually permit an elegant expression of our logic. In addition it is economical, in that, when one of the tests within the statement is satisfied and the corresponding action undertaken, execution of the conditional statement terminates.

The use of braces to signify a compound statement adds considerably to the expressive power of the conditional statement, in that the execution of groups of statements can be made dependent on a specific condition. This can perhaps be appreciated in example 3.2 where the main part of the program is executed only if the output file is opened satisfactorily.

Perhaps the part of the conditional statement that it is most important to understand is the condition itself. The general form of the statement showed this to be an expression enclosed by parentheses. Expressions will be considered in greater detail in chapter 6. For the present we can use the comparison of simple data items as an example of the form of expression required. An expression such as

$$n > 7$$

can be evaluated as soon as n is known. We expect the result 'true' if n is greater than 7 and 'false' otherwise. Convention dictates that we regard the value zero as 'false' and non-zero as 'true'. Thus, if the parenthesised expression following *if* yields a non-zero or 'true' value the statement that immediately follows is executed, and the *else* part, if it exists, is ignored. However, if the parenthesised expression yields a zero or 'false' value, the statement that follows *else* is executed. This property is exploited in the following function

```
    /* to determine whether `ch' is */
    /* the letter 'y', or 'Y'       */

affirmative(ch)
    char ch;
    {
        if ( ch=='y' ) return(1);
        else if ( ch=='Y' ) return(1); else return(0);
    }
```

If the character passed to *affirmative* is an upper case or lower case 'y' the value 1 is returned, otherwise 0 is returned. Such a function can significantly help the readability of our program. For, after prompting the user for a single character reply 'reply' to a question, we could then write

```
if ( affirmative(reply) ) printf("reply is yes\n");
```

Note that it is not necessary to compare the value returned by *affirmative* with zero or anything else. Indeed to do so would detract from the readability of the resulting statement. We could of course exploit the same principle by writing

```
if ( n ) printf("n is non-zero\n");
```

but we would argue that this is not good practice as n represents numeric values rather than the 'true' or 'false' values that *affirmative* represents.

(For illustrative purposes, the body of *affirmative* is more verbose than it need be. This function would normally be written in C as

```
affirmative(ch)
    char ch;
    { return(ch=='y' || ch=='Y') }
```

where | | is the 'or' operator.)

TRAPS FOR THE UNWARY

Consider the two statements

```
if ( ch='Y' ) return(1);
if ( ch=='Y' ) return(1);
```

and ask whether you can clearly state what each does. They differ only in that the first has one less 'equals' sign than the second. There is, nonetheless, a significant

difference in their actions. The second statement tests whether 'ch' has the value 'Y', returns 1 if it does and continues with the next statement in sequence if it does not. In contrast the first statement assigns the value 'Y' to 'ch' then, because an assignment is an expression that yields as its result the value assigned, the *return* statement is executed, since the parenthesised expression yields a non-zero value. This difference in action can be extremely important. Its advantage is that an assignment and a test of the assigned value are neatly combined. Its disadvantage is that if you intended comparison (= =) rather than assignment (=) your program is logically incorrect but syntactically correct. Those people moving to C from a language in which the single 'equals' sign is used for comparison are advised to check their conditional statements carefully.

In RatC there are several functions that use conditional statements in a simple but effective way. The functions *an*, to determine whether a character is alpha-numeric, and *alpha*, to test for an alphabetic character, are useful examples.

MULTIPLE CONDITIONS

Let us assume that we are given an integer, which is an examination mark, and that we are to translate this mark into a grade. An A grade is obtained for a mark in the range 80 to 99, B for a mark in the range 60 to 79, and so on. The character NULL is returned for a mark outside the range 0 to 99. There is, as usual, more than one way to achieve this end, but a look at several methods will help to contrast the use of different facilities in C.

Example 4.2

```
grade(mark)
    int mark;
    {
      char g;

      if ( mark<0 ) g=NULL;
      else if ( mark<20 ) g='E';
      else if ( mark<40 ) g='D';
      else if ( mark<60 ) g='C';
      else if ( mark<80 ) g='B';
      else if ( mark<100 ) g='A';
      else g=NULL;

      return(g);
    }

[ style 49.3 ]
```

While the logic of the statement is simple and economical, it is lengthy. What is needed to deal with the problem of example 4.2 is a construct that offers a multiple choice of actions in contrast to the binary choice offered by the conditional statement. The *switch* statement is just such a construct.

THE *switch* STATEMENT

The general form of the switch statement is

switch (expression) statement

The value yielded by the expression must be of type *int* (or *char* since the conversion to *int* is automatic) and will be used to select which of several statements to execute. The statement that follows the selecting expression will, if the switch is to serve any useful purpose, contain one or more statements preceded by

case constant_expression:

The constant expression can be thought of as labelling the statement that it prefixes. This statement is executed if the selecting expression yields a value that matches the constant expression. Within any *switch* statement the constant expression that labels a statement must be unique. A rewritten version of the mark grading example should make clear the form and logic of the *switch* statement.

Example 4.3

```
grade(mark)
    int mark;
    {
      char g;

      switch (mark/20)
          {
            case 0: g='E'; break;
            case 1: g='D'; break;
            case 2: g='C'; break;
            case 3: g='B'; break;
            case 4: g='A'; break;
          }

      return(g);
    }


[ style 40.5 ]
```

The unexpected feature of this example is, perhaps, the *break* statement. When it is encountered it causes exit from the *switch*. If in the example 4.3 the first *break* were omitted, then having assigned 'E' to 'g' the next statement, which assigns 'D' to 'g', is executed. In other circumstances, as we shall see, we might wish to exploit this course of action. It is not appropriate to do so in this example – all the *break* statements, with the exception of the last, are essential.

Example 4.3 is logically similar to example 4.2. It is not identical in its action, as NULL is not returned if 'mark' is outside the expected range. A statement prefixed by *default* is executed if the value produced by the switching expression does not match any of the constants following *case* within the switch statement. In example 4.3 when none of the *case* constants is matched exit is made from the *switch* statement. We can ensure that marks which are out of range are satisfactorily processed by including the statement

```
default: g=NULL; break;
```

anywhere within the *switch* statement of example 4.3. Finally we note that no ordering of the *case* or *default* prefixes is necessary or implied. The example 4.4 should make these points clear.

Example 4.4

```
        /* to determine whether a given character */
        /* is a vowel. Zero is returned for non-  */
        /* vowels. An integer in the range 1 to 5 */
        /* is returned for a vowel.               */

vowel(ch)
    char ch;
    {
      switch (ch)
          {
            default: return(0);
            case 'u': case 'U': return(5);
            case 'a': case 'A': return(1);
            case 'e': case 'E': return(2);
            case 'i': case 'I': return(3);
            case 'o': case 'O': return(4);
          }
    }


[ style 49.2 ]
```

This example exploits the fact that a *case* which is not followed by a *break* causes the following statement to be executed. In this way we can easily deal with both

upper and lower case versions of the characters. The statement prefixed by *default* could as easily be the last statement of the switch as the first. Another feature exploited is the use of *return* rather than a *break* statement. *return* causes exit from the *switch* statement and from the function.

The RatC compiler does not support a *switch* statement and therefore none is used in the RatC program. The function *statement* within RatC has the task of determining what kind of statement is about to be processed. It uses a conditional statement of the form illustrated in example 4.2. An integer indicating which kind of statement was detected is returned to the caller.

SUMMARY

In this chapter we have discussed two of the constructs that give programming its flexibility - the two-way and multi-way branch. Strictly, from the point of view of the logic of a program, one of the constructs is unnecessary, since either can be expressed in terms of the other. Careful use of the appropriate construct can, however, considerably enhance the intelligibility of a program.

A two-way branch will almost always be implemented with a conditional statement; a multi-way branch can be implemented either by nested conditionals or by a *switch* statement. As a general rule, we can say that nested conditional statements should be used whenever we are testing a series of conditions in decreasing order of expected frequency; when all the conditions are equally likely to occur, a *switch* statement should be used.

5 Loops

The conditional statements of the previous chapter freed our programs from the straitjacket of the sequential execution of instructions without branching, but it is the ability to loop, or repeat the execution of one or more instructions, that brings power to programming. It brings economy too, for a modest number of programming language statements can be responsible for a significant amount of computing time.

C offers at least three ways in which we can construct loops. We can use a *while* statement, a *do* statement, or a *for* statement. Of these, the *while* statement is the most important, because it can be used to do anything that the other two loop constructs can do. The other two forms of loop construct are available because, in certain circumstances, they offer a more appropriate means of expressing our logic. The RatC processor only offers the *while* statement as a looping construct and thus that is the only form of loop used throughout RatC.

THE *while* STATEMENT

The *while* statement has the general form

    while (expression) statement

The list of statements given at the start of chapter 4 must now be extended to include the *while* statement. Any one of this extended list of statements is admissible as the statement part of the general form of the *while* statement given above. The expression in parentheses has the same role as the parenthesised expression of the conditional statement – that is, it is evaluated and tested. If it produces a non-zero or 'true' result, the statement that follows is executed. The expression is then tested again and, if 'true', the statement following is executed once more. This sequence is repeated until the evaluation of the expression yields a 'false' result, and then the statement that follows the *while* statement is executed.

There is, of course, an implicit assumption that something occurs within the *while* loop which causes the value produced by the controlling expression to change at some time. The statement

```
while (1) i=0;
```

causes an infinite loop, setting 'i' to zero interminably. Care must be taken to ensure that loops do terminate!

In example 5.1 we introduce two new operators, !=, and ++. The first tests for inequality; the second is the increment operator, which when used as in

```
count++;
```

causes 'count' to be incremented by one. Suppose our task is to count the number of characters on a line. Assuming that the input stream is positioned at the start of a line, the following statements perform the count

```
count=0;
ch=getchar();

while ( ch!='\n' )
    {
      count++;
      ch=getchar();
    }
```

But these statements do not exploit some of the features that we have already seen. In particular, the test that controls the *while* statement could easily be modified to include the assignment to 'ch'. The modified version uses this feature and is presented as a function.

Example 5.1

```
counter()
    { char ch;
      int count=0;

      while ( (ch=getchar()) != '\n') count++;
      return(count);
    }


[ style 46.6 ]
```

Example 5.1 also capitalises upon the ability, in C, to initialise variables as part of their definition. A closer look at the function *counter* should prompt the realisation that 'ch' is used only in the expression that controls the *while* loop. If this is so, then we should dispense with 'ch' altogether and rewrite the function as in example 5.2.

Example 5.2

```
counter()
    { int count=0;

      while ( getchar() != '\n') count++;
      return(count);
    }


[ style 48.4 ]
```

In this, and other ways, C offers many aids to writing 'economical' (some would say terse) programs. The reader is encouraged to exploit these features but to bear in mind that simplicity and clarity of expression should not be sacrificed in order to produce 'smart', but not easily readable, programs.

ESCAPING FROM LOOPS

The *break* statement, which was used to escape from the *switch* statement, will also force exit from a *while* statement. Following the execution of *break*, the statement that follows the *while* statement is executed. A *return* statement also may be used to escape from a *while* loop. However, as might be expected, this not only causes immediate exit from the *while* statement, but also forces exit from the function that contains the *while* statement. The function *doasm* of RatC uses the *break* statement to escape from a *while* loop. *doasm* is invoked when the directive indicating that 'assembly language statements follow' is discovered. All lines of input that contain assembly language statements are simply copied to the output file until either 'end of file' or the terminating directive is discovered.

The *while* statement can also be exploited when attempting to make the user interface of a program more robust. If a program directs a query to its user which requires a simple 'yes' or 'no' answer, for example

Do you wish to continue (Y or N) ?

then only the response indicated should be accepted. Consider example 5.3.

Example 5.3

```
#define BELL '\7'

replyisyes()
    { char ch;

      while (1)
```

```
        {
          ch=getchar();
          switch (ch)
              {
                default: putchar(BELL); break;
                case 'y': case 'Y': return(1);
                case 'n': case 'N': return(0);
              }
        }
}
```

```
[ style 61.3 ]
```

Exit is only made from the function when 'Y' or 'N' of either upper or lower case is received. Receipt of any other character causes the VDU to 'beep' and, although exit is made from the *switch* statement, the *while* statement remains active.

This last example provides the opportunity to state again that a program's interface with its user is extremely important. If a question is directed to the user, ensure that the acceptable responses are made known, and write the program logic in such a way that only valid responses are accepted.

Further details of the input/output philosophy of the underlying Operating System will need to be clarified before example 5.3 can be used conveniently. Usually, for example, a user is required to provide 'line at a time' input. That is, a character followed by 'newline' would be expected. Example 5.3 would 'beep' at any 'newline' character that it encountered. It is usually possible to arrange 'character at a time' input, but the mechanism for achieving this will be environment dependent and thus outside the scope of this book.

The *while* loop is important because, as is evident from its structure, the controlling condition is tested before entering the loop. In contrast, the expression that controls the *do* loop is tested only at the end of the loop, and therefore the statement controlled by the loop is always executed at least once.

THE *do* STATEMENT

The general form of the *do* loop is

do statement while (expression)

Our list of statements must now be extended to include the *do* statement. Any one of the resulting list of statements is suitable as the statement used in the general form given above.

As an illustrative example, let us assume that we have access to a file containing one word per line. Our task is to sum, for each such word, the number of times that we find a vowel preceded by a consonant. The sum produced is a good approxima-

tion to the number of syllables in the word. We assume a file pointer 'fptr', and a function *consonant* which returns a non-zero (true) value if the character passed as a parameter is a consonant. The function *vowel* was given as example 4.4.

Example 5.4

```
syllables()
    { char ch;
      int changes=0, previousvowel=0;

      do
          {
            ch=getc(fptr);
            if ( vowel(ch) )
                {
                  if ( !previousvowel ) changes++;
                  previousvowel=1;
                }
            else if ( consonant(ch) ) previousvowel=0;
          }
      while ( ch != '\n' );
      return(changes);
    }

[ style 52.7 ]
```

(As a syllable counter, the function of example 5.4 is limited in that there are special cases that it does not handle. Thus 'by' would be credited with having no syllables, and 'ale' with two. For most words, however, it is a good first approximation.)

THE *for* STATEMENT

The *for* statement proves convenient to use when it is necessary to execute a loop a given number of times. While this could also be done by either of the other two loop constructs, we should select the statement that is most appropriate for the task. Counting through a loop requires three 'housekeeping' activities: initialising the counter, incrementing the counter, and testing whether the terminating value has been reached. It is helpful to both the reader and the writer of a program if these three housekeeping activities are collected together. This is economically achieved in the *for* statement which has the general form

for (expression1; expression2; expression3) statement

where

expression1	initialises the counter,
expression2	gives the continuing condition, and
expression3	increments the counter.

Thus to compute the sum of the first N natural numbers we could write

```
sum=0;
for ( i=1; i<=N; i++ ) sum=sum+i;
```

or, if it is more suitable to count down

```
sum=0;
for ( i=N; i>=1; i-- ) sum=sum+i;
```

In C, the statement controlled by the *for* statement in these examples can be more concisely written as

```
sum+=i;
```

THE *continue* STATEMENT

We have seen that *break* will cause immediate exit from a *switch* or *while* statement. It will also cause immediate exit from a *do* statement or *for* statement. The loop statements (*while*, *do*, and *for*) can also use a *continue* statement. The *continue* statement is less drastic than the *break* statement because it only causes termination of the present iteration. If *continue* is encountered in the execution of *while* or *do* loops, it causes a branch to the loop control test to be made. In a *for* statement a *continue* causes execution of the 'increment' expression prior to testing whether another iteration of the loop is appropriate.

Imagine that a file contains a collection of marks, except that the very first number in the file gives the number of marks that follow. Using the function *grade* of example 4.2, we are to compute the number of pass grades in the mark list (example 5.5).

Example 5.5

```
        /* n.b. fscanf may return EOF or zero;   */
        /*      grade returns NULL if the        */
        /*            mark is out of range;      */
        /*      only an E grade does not pass.   */

passes()
```

```
{ char g;
  int  listsize, mark, m, psum=0;

  if ( fscanf(fptr, "%d", &listsize) < 1 ) return(-1);

  for ( m=1; m<=listsize; m++)
      {
        if (fscanf(fptr, "%d", &mark) > 0 )
            {
              if ((g=grade(mark)) == NULL) continue;
              if (g=='E') continue;
              psum++;
            }
        else return(-1);
      }
  return(psum);
}
```

```
[ style 65.9 ]
```

DYNAMIC CHANGE OF INCREMENT

The *for* statement in C is implemented in a manner that enables it to be used in some rather surprising ways. For example

```
for ( ; ; ) k=0;
```

represents an infinite loop. The assumption is made that, if the second expression, which is the controlling condition, is omitted, the value 'true' is to be used. The most significant way that the *for* statement differs from the *for* statement as defined in, say Pascal, is that both the terminating condition and the increment expression are re-evaluated for every iteration. This means that if the identifiers used in computing these values are changed within the *for* loop, then either the terminating condition, or the step size, or both, can be constantly changed from within the loop. Consider, for example

```
for ( p=1; p<=4096; p=2*p) printf("%4d\n", p);
```

which prints a small list of powers of two. It achieves this by multiplying the 'increment' by two each time through the loop.

The loop terminating condition need not involve the 'counter', although it usually will. The loop of example 5.2 could be rewritten, using a *for* statement, in the following form

```
for ( count=0; getchar() != '\n'; count++ );
```

Here the *for* statement has an empty statement part, because all the necessary work is done within the controlling expressions. Note that the terminating condition is independent of 'count'. Changing the loop terminating condition from within the loop should be done carefully, if at all. There is a danger that it may be changed in such a way as to ensure that the loop never terminates at all.

A final example on *for* statements is used to show that they, or any of the other looping constructs, may be nested to create a loop within a loop. Example 5.6 computes 'perfect' numbers. If we exclude the number itself from a list of its factors, then a perfect number is the same as the sum of its factors, so that the first perfect number is 6, because the factors of 6 are 1, 2 and 3, and 1+2+3 = 6. It is only necessary to examine even numbers for perfection, because, although it remains to be formally proved, it is surmised that odd numbers cannot be perfect.

Example 5.6

```
#define LO   6          /* first perfect number */
#define HI   1000       /* limit of search      */

main()
    { int num, sum, factor;

      printf(" Perfect numbers \n");
      for ( num=LO; num<=HI; num+=2 )
          {
            sum=1;
            for ( factor=2; factor<num; factor++ )
                if ( num%factor == 0 ) sum+=factor;
            if ( sum == num ) printf("%4d\n", num);
          }
    }

[ style 63.8 ]
```

The modulus operator, %, is described in more detail in chapter 6. It gives, in this case, the remainder when 'num' is divided by 'factor'.

THE *goto* STATEMENT

The loop structures introduced so far, if used properly, should mean that the user rarely, if ever, needs to use a *goto* statement. In particular, a *goto* need never be used to construct loops. However, in certain error situations, a *goto* may enable a cleaner

program termination to take place. A statement may be labelled by prefixing it by an identifier followed by a colon. The *goto* statement may then use this label as its destination, thus

```
goto abort;

   ....
abort: printf(" abnormal termination \n");
```

SUMMARY

C's looping constructs correspond to those found in many other high-level languages. Usually, a determinate loop, where the number of iterations is known in advance, is most appropriately implemented by a *for* statement, while an indeterminate loop, where termination depends on some condition being satisfied, is better implemented as a *while* or a *do* statement. These are general rules, however and, as has already been demonstrated, C's *for* statement is powerful enough to enable it to be effectively used to control an indeterminate loop under certain circumstances. This being so, it is wise to consider carefully which particular statement is likely to yield the most natural expression of the loop's intent.

6 Operators

In preceding chapters we have used identifiers with type *char*, *int*, and *float*. Data types *char* and *int* must be available in any C implementation. When the size of a C language processor has to be reduced, it will be the data type *float* that will be sacrificed. RatC does not support the type *float*. On larger machines offering a 'full' implementation of C we might also expect to have access to the types double length floating point (abbreviated to *double*), double length integer (abbreviated to *long*) and perhaps short integers (*short*). We suggest that you look at the implementation notes for C on your system to discover what is on offer. This information is needed only when it is necessary to mix types in an expression, for then we need to know the type of the result.

TYPE CONVERSION

The type names introduced above can conveniently be listed in order as follows

char, short, int, long, float, double

Apart from the *long/float* boundary, this list is in order of increasing storage size. By storage size we mean the amount of storage needed for a data item of the given type. With this list in mind the implicit type conversion rules given below can readily be understood.

For an expression involving one of the binary operators (one with two operands), such as

```
a + b
```

the type of the result is determined by the type of the operands according to the following rules. *char* and *short* are converted to *int*, and *float* is converted to *double*. If, as a result, either operand ('a' or 'b') is of type *double*, the other is converted to *double*. As a result of this conversion either both operands are *double*, in which case the result is *double*, or one or both of them is *int* or *long*. If either operand is *long* the other is converted to *long* and the result is *long*, or they are both *int* and the result is *int*. The implicit conversion is therefore always from the 'smaller' object to the 'larger'. The results of type conversion are summarised in table 6.1. An explicit type conversion can be obtained by using a 'cast'.

Table 6.1

a	b	Result
char short int	char short int	int
char short int	long	long
char short int	float double	double
long	long	long
long	float double	double
float double	float double	double

CAST

By prefixing an expression with one of the type names used earlier enclosed in parentheses, we force the expression to yield a result of the type indicated so that

```
(long) 2+3
```

produces the result 5 which has type *long*. A cast can also be useful in forcing an actual parameter to have the type of the corresponding formal parameter. The functions *exp*, *log*, and *sqrt*, which are to be found in the library of mathematical functions, expect a parameter of type *double*, and produce a result of type *double*. If we wish to obtain the natural logarithm of 'x', which has type *float*, then we can write

```
log( (double) x )
```

The assignment operator is treated in a different way to most of the other operators. The type of the expression of the right-hand side (rhs) is changed to the type of the identifier on the left-hand side. In appropriate circumstances, therefore, a rhs of type *double* is rounded to *float*, a rhs with type *float* is truncated to *int*, and an *int* is converted to *char* by ignoring excess high order bits.

ASSIGNMENT OPERATORS

We have introduced a limited number of these operators at suitable places in the text. For example, the operator += was used to enable us to write

```
sum+=i;
```

rather than

```
sum=sum+i;
```

An assignment in C is treated like any other operator in that, having made the assignment the value assigned is available for other use. Thus

```
(sum+=i) > max
```

adds 'i' to sum and compares the assigned value with 'max'. The validity of a 'multiple assignment' should therefore be apparent.

```
sum=total=start=0;
```

The full list of assignment operators is

```
+=          -=
*=          /=
%=          >>=
<<=         &=
^=          |=
```

The meanings of the various assignments will become obvious as we consider the different groups of operators.

ARITHMETIC OPERATORS

We will introduce operators in the various groups by using them in simple expressions. While this may not be strictly necessary for the more familiar operators, it should help to clarify the action of the less familiar ones.

```
  - 5     -5    (unary minus)
7 + 5     12     add
7 - 5      2     subtract
7 * 5     35     multiply
7 / 5      1     divide
7 % 5      2     modulus
```

With the exception of the modulus operator, the type of the result from such expressions will, in general, be determined by the conversion rules given earlier. In the examples above, all results are of type *int*. When two items of type *int* are divided, the fractional part of the result is truncated to produce a result of type *int*. The modulus operator produces the remainder after division of one integer by another. The result is of type *int*. Operands of type *double* or *float* may not be used with this operator.

A small example which uses most of the operators above is a function to evaluate Zeller's Congruence (Uspensky and Heaslet, 1939), shown in example 6.1. This function, when given a day, month, and year (full form), produces a result in the range 0 to 6. With Sunday as day 0, this number represents the day of the week on which the given date fell. It can be used, for example, to determine birthdays.

Example 6.1

```
/* zeller returns a number in the range  */
/* 0..6 representing the day of the week */
/* on which the given date falls.        */
/* Sunday is day 0.                      */

zeller(day,month,year)
    int day, month, year;
    { int temp, yr1, yr2;

      if (month<3) { month+=10; year-=1; }
      else month-=2;

      yr1=year/100; yr2=year%100;
      temp=(26*month-1)/10;

      return((day+temp+yr2+yr2/4+yr1/4-2*yr1+49)%7);
    }

[ style 51.3 ]
```

BITWISE OPERATORS

C enjoys well-deserved popularity as an 'implementation' language. This is in large measure due to the ease with which the user can access and manipulate bit patterns in memory. The following operators are available

7 << 5	224 (0xE0)		left shift
7 >> 5	0	*	right shift
7 | 5	7		inclusive or
7 ^ 5	2		exclusive or
7 & 5	5		and
~05	0177772		one's complement
		*	beware sign propogation

Note the use of hexadecimal and octal constants above – hexadecimal constants are written with a leading *0x* or *0X*, and may use digits *0* through *9* and letters *A* through *F* (or *a* through *f*); octal constants are written with a leading *0*, and may use digits *0* through *7*. The last example, of the one's complement operator, assumes that the length of an *int* is 16 bits.

Bit manipulation, usually the preserve of assembly language programmers, is necessary, for example, when checking the bits of a status register and in masking data to be received or transmitted. An example to illustrate use of these operators need not be drawn from such a machine specific area. The 'feedback shift register' technique for generating pseudo-random numbers is easily expressed using the bitwise operators as example 6.2 shows.

Example 6.2

```
#define MAXINT 32767
#define PSHIFT 4
#define QSHIFT 11

random(range)
    int range;
    { static int n=1;

      n=n^n>>PSHIFT;
      n=(n^n<<QSHIFT)&MAXINT;
      return(n%(range+1));
    }

/* the function is dependent upon */
/* the word length of the host    */
/* machine. The seed `n' should   */
/* be capable of easier change    */
/* than is possible here.         */

[ style 64.1 ]
```

The rationale behind this algorithm, which is a good source of random numbers, is given in Lewis (1975). A Pascal version, which makes an interesting comparison, is given in Meekings (1978). Remember too that since C makes it easy to print the value of a variable in either octal or hexadecimal, the results of bitwise operations can usually be displayed in an easily assimilated form.

LOGICAL OPERATORS

These operators are usually used to combine one or more comparisons in the controlling expressions of conditional statements, *while* statements, and the other loop constructs.

```
7 && 5    1     logical and
7 || 0    1     logical or
   ! 0    1     logical not
```

The important point that distinguishes these operators from the bitwise operators is that any non-zero operand is treated as 1 (true). A zero operand is treated as false. The result of the operation is 0 or 1 according to the normal rules for logical connectives. Expressions using && and || are evaluated left to right and evaluation should terminate once the truth or falsity of the expression is determined. None of these operators is used in RatC, and none is processed by RatC. In consequence some of the expressions that would normally be written using the logical connectives are written using several conditional statements to obtain the required termination as soon as possible. The group of functions in RatC that deal with expression processing, *hier1* to *hier11*, contains several examples of such constructions. For illustrative purposes, imagine that we wish to compute the mean rainfall given the total rainfall 'train' over a number of days 'days'. We might write

```
if (days>0)
    if ( (mean=train/days) > 5.0) print("%d\n", mean);
```

assuming that we wished to avoid division by zero. But consider

```
if ( (days>0) && ((mean=train/days) > 5.0))
```

as an alternative test. It is only a useful alternative if, when 'days' is zero, the expression in which 'days' is a divisor is not evaluated. C guarantees that when the truth or falsity of an expression is known, as it is above when (days > 0) evaluates to zero (false), evaluation of the expression immediately terminates.

RELATIONAL OPERATORS

Examples of some of these operators have appeared at several places in the text so far. The operators are

```
>    greater than
>=   greater than or equal to
==   equal
!=   not equal
<=   less than or equal
<    less than
```

The test for a digit is a simple example of the use of two relational operators and a logical operator

```
digit = ( ch >= '0') && ( ch <= '9' );
```

INCREMENT AND DECREMENT

The usefulness of the increment operator should by now have become apparent. The decrement operator is used in an entirely similar fashion, so that

```
countdown--;
```

decrements countdown by one. What has not been emphasised so far is that both the increment operator and the decrement operator may be used either as a prefix or postfix to an operand. We may therefore write

```
++count;          --countdown;
```

Such simple usage as this does not make clear what difference there might be between the prefixed or postfixed operator. The difference can be illustrated by the following example

```
up=0;
printf("%2d\n", up++);     /* prints 0 */
printf("%2d\n", ++up);     /* prints 2 */
```

The first statement after the initialisation will print zero and then increment 'up'. In the second print statement the value of 'up' will be incremented (to two) and then printed. The prefixed form means increment (or decrement) and use, while the postfixed form means use and then increment (or decrement). The difference is important, as we will see, when dealing with array subscripts.

CONDITIONAL OPERATOR

The conditional operator affords an easy and compact way to express a value which depends on a test. In the following example, the absolute value of x is computed.

```
if (x < 0)
        xabs = -x;
else
        xabs = x;
```

C gives us a more concise way to write such things, so their meaning becomes more apparent. The conditional operator takes three expressions and is used in the following format

expression-1 ? expression-2 : expression-3

Expression-1 is evaluated and then tested. Based upon the results of this test, either one (but not both) of expression-2 and expression-3 will then be evaluated and that value will become the result of the whole conditional expression. If the value of expression-1 is true (non-zero), expression-2 is evaluated; otherwise, expression-3 is evaluated. Thus, we can write the absolute value computation as

```
xabs = (x<0) ? x : -x;
```

Printing a heading only after a certain number of lines suddenly becomes easy to write

```
#define HEADING         "\n\n\n      - Treasure Island -\n\n\n"

printf ("%s", (no_lines % 60 == 0) ? HEADING : "");
```

Standard conversion rules will be used to bring the constituent values of the conditional expression to a common type to produce the result. So, in the following example, if x is of type float when it is substituted by the preprocessor, the resulting type of the whole conditional expression is a float.

```
#define min_1(x)        (x>1 ? x : 1)
```

COMMA OPERATOR

The comma operator is syntactic sugar: it need not be provided since there are other facilities in the C language which can accomplish the same function; its use is more a question of style than of functionality. Expressions connected by a comma operator are executed in sequence. One use might be to initialise several quantities in a for statement. The following code might be used to scramble the letters in a word five successive times

```
for (count = 0, j = word; count++ < 5; j = scramble (j))
        ;
```

First the expression on the left of the comma is evaluated and the result discarded; then the expression on the right of the comma is evaluated and used as the resulting value. The type of the result is the type of the operand on the right of the comma.

Ambiguity can arise in the cases where the comma can also be interpreted as a character separating items in a list (that is, arguments and initialisers). In those circumstances, the comma operator can only be used inside parentheses

```
my_func (arg1, (c = C_INIT, (c + 1)*10), arg3);
```

PRECEDENCE OF OPERATORS

Whatever programming language you use it is important to write expressions in a way that makes sense to you, the writer. (Bear in mind too that others will wish to read and understand your program.) In order to do this, and still produce programs that are syntactically and logically correct, it is necessary to understand how expressions are written and how they are interpreted. Operands must be separated by operators, and evaluation usually proceeds from left to right. Thus, in an expression such as

```
a + b * c
```

it can be seen that the operators separate the operands, but we are accustomed to the multiplication of 'b' and 'c' being carried out before the addition of 'a'. Formally we say that multiplication has a higher priority or precedence than addition. Parentheses can always be used to enforce the required priority. In C, however, there are occasions on which even this rule may not be as easy to apply as we would wish. Another possible source of confusion is that some operators, for example * and &, have more than one role. Consider for example

```
*pint++
```

which is not part of a multiplication. It might mean increment the pointer (address) 'pint' by one and retrieve the contents, or it might mean that the value '*pint' is to be increased by one. In fact unary operators are evaluated from right to left and so the expression increments the pointer 'pint' and not what it points to. The latter effect is achieved by

```
(*pint)++
```

It is therefore important to know the order of precedence of operators and the direction of association. A table of this information is given in table 6.2. Operators

are listed in decreasing priority, with operators in the same section having equal priority.

SUMMARY

C has a well-deserved popularity among high-level and low-level programmers alike. Such popularity is, in large part, attributable to the richness of its set of operators, which allows a clear and natural expression of the program logic, with the additional bonus of an efficient translation into the underlying machine instructions. It is the large variety of operators that characterise the language, and possibly pose the greatest hurdle for the novice C programmer.

Time spent initially in learning how to use the full set of operators will be amply rewarded by clear, concise and efficient programs.

Table 6.2

Operator	Name	Associativity
( )	parentheses	left to right
[ ]	brackets	
–>	pointer	
.	dot	
++	increment	right to left
- -	decrement	
(type)	cast	
*	contents of	
&	address of	
-	unary minus	
~	one's complement	
!	logical NOT	
sizeof	size of	
*	multiply	left to right
/	divide	
%	modulus	
+	plus	left to right
-	minus	
>>	shift right	left to right
<<	shift left	
>	greater than	left to right
>=	greater than or equal	

Operator	Name	Associativity
<=	less than or equal	
<	less than	
==	equal	left to right
!=	not equal	
&	bitwise AND	left to right
^	bitwise exclusive OR	left to right
|	bitwise inclusive OR	left to right
&&	logical AND	left to right
||	logical OR	left to right
?:	conditional	right to left
=	equals	right to left
+=	plus equals	
-=	minus equals	
*=	multiply equals	
/=	divide equals	
%=	modulus equals	
>>=	shift right equals	
<<=	shift left equals	
&=	and equals	
^=	exclusive or equals	
|=	inclusive or equals	
,	comma	left to right

7 Arrays

In the examples used so far each data item that we wished to manipulate has been given a name, or identifier. Each identifier has associated with it a type, and a storage class. This association is made explicit through the declaration. But so far any identifier has represented a numeric value of one type or another, or a character. Consider again example 4.3 in which we produced a grade for a given mark. If we now change the specification of the problem, to ask that we produce the number of times that each grade was achieved, the statements in example 7.1 could appear in a suitable loop.

Example 7.1

```
/* assume a=b=c=d=e=f=0; prior to loop entry */

switch (mark/20)
    {
      case  0: e++; break;
      case  1: d++; break;
      case  2: c++; break;
      case  3: b++; break;
      case  4: a++; break;
      default: f++;
    }
```

While we can contemplate writing this when only five grades are involved, we would, if twenty-five grades were involved, look for a 'better way'.

ARRAY DECLARATIONS

Instead of having individual identifiers for each grade total, which causes difficulty when dealing with them collectively, what would be much more useful would be a collective name for the grade totals together with a method of accessing each grade total. A street name is a collective name for several houses. The house number uniquely identifies each house of the street. An array name is a collective name for several data items of the same type. Each item has a unique reference number

known as an index or subscript. If 'grades' is the collective name for the five grade totals it could be declared as

```
int grades[5];
```

In C array subscripts start at zero. The five grades can therefore be referred to as

grades[0], grades[1], grades[2], grades[3], grades[4]

POINTERS AND ARRAYS

Another method of referring to the individual elements of an array is available to us in C. The array name, 'grades' in this case, is always treated as a pointer, or address. It points to the first element of the array. If, for example, we make a copy of the pointer, then we can increment and decrement the pointer value in order to refer to different elements of the array. Consider example 7.2.

Example 7.2

```
int grades[5], *gptr;
gptr=grades;     /* gptr points to grades[0] */
gptr++;          /* gptr points to grades[1] */
gptr++;          /* gptr points to grades[2] */
```

A subscript within square brackets is the more usual way to refer to elements within an array. Use of a pointer, while initially not so familiar, can become more convenient and is usually more economical in implementations of C. We shall move towards use of pointers for array access.

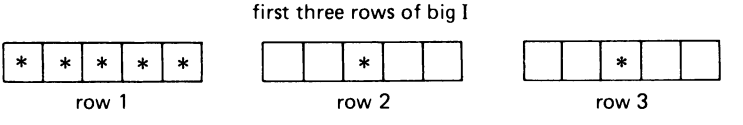
With an array to help us, we can now write example 7.1 in the following way

```
int grades[5], *gptr, s;

    /* initialise array elements */

    gptr=grades;
    for (s=0; s<5; s++) *gptr++=0;

    /* assume a function  getmark' which */
    /* returns either the next mark or   */
    /* -1 to indicate the end            */

    while ((mark=getmark()) != -1)
        {
          s=mark/20;
          if ( (s>=0) && (s<5) ) grades[s]++;
        }
```

There are several points of interest in this example. First note that the explicit constant 5, the number of elements in the array, appears three times in the program text. A symbolic name should be 'defined' to have this value, thus making a change in array size easy to accommodate. Secondly, note that the array elements are zeroised using the pointer 'gptr', and finally note that the increment operator can be used on an array element just as on any other variable.

ARRAYS OF MORE THAN ONE DIMENSION

C allows us to use arrays of more than one dimension. Imagine that instead of simply printing letters in a 7*5 grid, as we did in the early examples of chapter 2, we wish to store these representations of characters in a 7*5 array, that is, an array with 7 rows and 5 columns. If we wish to access these elements using a pointer, then it is essential to appreciate that in C arrays are stored by row.



This means that the rightmost of the two subscripts changes more quickly because elements are accessed in the order that they are stored. A two-dimensional array can easily be visualised as a table, and therefore we shall initially use subscripts, rather than a pointer, to access the elements (example 7.3). We shall later rethink this approach.

Example 7.3

```
#define ROWMAX 7
#define COLMAX 5

char letter[ROWMAX][COLMAX];
int  col;

    /* fill array with spaces */

for (row=0; row<ROWMAX; row++)
    for (col=0; col<COLMAX; col++) letter[row][col]=' ';

    /* alternatively we could write .. */

for (row=0; row<ROWMAX; row++)
    for (col=0; col<COLMAX; letter[row][col++]=' ');
```

Observe that each subscript is enclosed by square brackets and that the final *for* statement does not have a statement to control. This is because each element of 'letter' can be set to a space in such a way that the column subscript is incremented after it has been used to access the array element. This is an occasion where use of

++col rather than col++

would not have the required effect.

ARRAYS AS PARAMETERS

Pursuing our example a little further, for those upper case letters of the alphabet that can be constructed from horizontal and vertical lines only, it would be convenient to have functions that fill a row, or a column, with a given character. The functions of example 7.4 fulfil this task.

Example 7.4

```
/*                NB                    */
/* the following defines are assumed */

#define ROWMAX 7
#define COLMAX 5

fillrow(row, matrix)
    int row;
    char matrix[ROWMAX][COLMAX];
    { int c;

      for (c=0; c<COLMAX; matrix[row][c++]='*');
    }

fillcol(col, matrix)
    int col;
    char matrix[][COLMAX];
    { int r;

      for (r=0; r<ROWMAX; matrix[r++][col]='*');
    }

[ style 53.5 ]
```

Each of the functions must change the contents of the array and, as we saw in chapter 2, must therefore have access to the address of the data item to be changed.

But since the array name is the address of the first element, it can be used without modification as a parameter to a function. The functions of example 7.4 will access the contents of the array that is the actual parameter, and it should therefore be obvious that the purpose of the line

```
char matrix[ROWMAX][COLMAX];
```

in each function is simply to establish the type of the formal parameter 'matrix'. No storage allocation is performed. It may not be necessary, but it is not wrong, to give the size of each dimension. Given that arrays are stored in row-major order, the size in the first dimension may be omitted, as it has been in the function *fillcol* of example 7.4.

It should be apparent that the functions of 7.4 also make use of what we called implicit parameters in chapter 2. *fillrow* uses 'COLMAX' which, although its definition is a *define* statement, could as easily have been, say, a static variable of the file containing the functions. The functions are not 'self-contained' in the sense that the identifiers that they use do not all derive from either the parameter list or the local variable declarations. This is a common occurrence but worth emphasising. Assuming the definitions of 7.3 and 7.4 we can write

```
makeH(mat)
    char mat[ROWMAX][COLMAX];
    {
      fillcol(0, mat);
      fillcol(COLMAX, mat);
      fillrow(3, mat);
    }
```

and thereafter write

```
makeH(letter);
```

STRINGS

In the preceding section we used an array of characters and, because of the particular example chosen, all elements of the array were always used. But when we wish to deal with strings, which are stored as an array of characters, it is inefficient to assume that the string will occupy all elements of the array in which it is stored. We must expect that either the length of the string is stored along with it, or that the end of a string is denoted by a special character. C adopts the convention that the end of a string is denoted by the NULL character '\0'.

Example 7.5

```
#define WIDTH 80

char mess[WIDTH], *m;

mess[0]='h';
mess[1]='e';
mess[2]='l';
mess[3]='l';
mess[4]='o';
mess[5]='\0';
```

The rather laboured statements of example 7.5 cause six characters to be stored in 'mess'. Since the last character is NULL we can say that the array 'mess' holds a string. The string may be printed by any of the following statements.

```
    /* assume the assignment m=mess */
    /*      for each alternative      */

while (*m != NULL) putchar(*m++);

while ( ((m-mess) < WIDTH) && (*m != NULL)) putchar(*m++);

printf("%s\n", m);
```

The tedious parts of the above examples are those that deal with individual characters. While this may sometimes be necessary, we more usually wish to process the string as a whole. We have been accustomed to writing a string as a sequence of characters between double quotes thus

```
" C-ing is believing "
```

It is therefore not unreasonable to expect that we may assign a string to an identifier without the necessity of doing it character by character. We achieve this as follows

```
char *sptr;
    sptr=" C-ing is believing ";
```

From its declaration 'sptr' is a pointer to a character. In particular, after assignment, 'sptr' points to the first character of the string. It is important to note that the assignment does not copy the character string. The declaration of 'sptr' offers no storage space for characters. The string is stored somewhere, we know not where, except that we have in 'sptr' a pointer to the first character. This is usually sufficient. If, for some reason, it is necessary to copy the string into local storage, then

this must be done with a function such as *strcpy* which copies a string from one storage place to another. In example 7.5 when storing one character at a time in 'mess' we were responsible for ensuring that a NULL character followed the last useful character. When, as above, a string is assigned to a pointer, a NULL is automatically appended to the character sequence. Use of pointers to refer to a string is much the most common and convenient way of dealing with strings in C. Any functions provided by a C implementation to help process strings, compare strings, find the length of a string, find a character within a string, will require the user to pass pointers as parameters.

Those functions in RatC that access and use the symbol tables of necessity process strings. *addglb*, *addloc*, and *addmac*, are responsible for adding global symbols, local symbols, and macro symbols, respectively, to the appropriate tables. The organisation of the tables is simple rather than efficient and the functions *findglb*, *findloc*, and *findmac*, find symbols in the respective tables by means of a linear scan through all names in the table. The functions *streq*, *astreq*, *match*, and *amatch*, also deal with strings. String comparison is done by *streq*, and *match*, while the variants *astreq*, and *amatch* compare strings over a given number of characters.

ARRAYS OF POINTERS

A program that was designed to report a variety of error messages to its user might use the approach given in example 7.6.

Example 7.6

```
char *error[30];

/* error is an array of 30 pointers to char */

error[0]="not enough arguments";
error[1]="too many arguments";
error[2]="invalid argument";

/* etc., etc. */

/* to report error number `i' */

printf("*** %s ***\n", error[i]);
```

The patterns of asterisks held in 7*5 arrays of characters, while not especially useful, are easily visualised. Imagine therefore, that we wish to construct, and store in this form, representations of all upper case letters of the alphabet. If lptr[$i-1$] is to point to the representation of the ith letter, then we need the declaration

```
char (*lptr[26])[7][5];
```

This declaration says that 'lptr' is a 26 element array of pointers. The pointers point to 7*5 arrays of characters. If we wish to associate the eighth pointer with the eighth letter of the alphabet, H, we could do this easily by the statement

```
makeH(lptr[7]);
```

The preceding examples should have helped to clarify the way in which two-dimensional arrays can be used in C. But a moment's reflection will reveal that in order to store our upper case characters in this manner we would need storage space for 26*7*5 characters. Furthermore, each character needs to be placed in the correct element. This is certainly not making best use of the facilities available in C. Even in our earliest examples we recognised that it was worth having functions or *define* statements to deal with five stars, a middle star, and two end stars (example 2.3). Following this course we could set up strings as follows

```
char *allstars, *endstars, *midstars;

    allstars="*****";
    endstars="*   *";
    midstars="  *  ";
```

An array of seven elements, where each element is a pointer such as 'allstars', can now be used to represent a character composed of asterisks. Thus the character H can now be represented by seven pointers, six of which point to the same object.

```
makeH(sptr)
    char *sptr[ROWMAX];

    {
      sptr++=sptr++=sptr++=endstars;
      sptr++=allstars;
      sptr++=sptr++=sptr=endstars;
    }
```

We now need an array of 26 pointers in which each pointer points to an array of seven pointers which point to strings. This is obtained with the declaration

```
char (*lptr[ROWMAX])[26];
```

The call to our new version of *makeH* defined above would be

```
makeH(lptr[7]);
```

The advantage of rethinking our example, or rather the way to express it in C, has been that we have eliminated the need to assign characters to individual array

elements. We now assign strings to pointers. Further, our storage requirement is considerably reduced as we store only one copy of each string (row) of characters. Each 'big' character can be represented by seven pointers and we need twenty-six such characters. We therefore save ourselves writing effort, storage space, and run time, by thinking about our task in a way which enables us to take full advantage of the facilities offered by C.

It is important, and useful, to be thoroughly familiar with the handling of strings and pointers in C. The next example, which is complete, should help to consolidate the work on strings.

Example 7.7

```
/* Soundex code generator: to transform a string */
/* into a code that tends to bring together all  */
/* variants of the same name (usually surname).  */
/*     -  (Knuth, 1973)                          */

main()
    { char    str[20];

      printf("\nCharacter string ? ");      /* ask user ..      */
      scanf("%s",str);                      /* for a string     */

      encode(str);                          /* encode all but   */
                                            /* the first char   */

      dumpdups(str);                        /* erase adjacent   */
                                            /* duplicate codes  */

      dumpzeros(str);                       /* erase zero codes */

      fixup(str);                           /* pad or truncate  */
                                            /* to four digits   */

      printf("\nSoundex code is  : %s\n",str);     /* tell user */
    }


encode(s)
    char    *s;
    { static char code[]="01230120022455012623010202";

      while (*++s) *s=code[*s-'a'];
    }
```

```
dumpdups(s)
    char    *s;
    { char    *t;

      while (*s)
          if (*s==*(s+1))
              { t=s+1; while (*t= *(t+1)) t++; }
          else s++;
    }

dumpzeros(s)
    char    *s;
    { char    *t;

      while (*s)
          if (*s=='0')
              { t=s; while (*t= *(t+1)) t++; }
          else s++;
    }

fixup(s)
    char    *s;
    { int     i;

      for (i=1; *++s && i<4; i++);
      for ( ; i<4; i++) *s++ ='0';
      *s=(char)0;
    }

[ style 55.7 ]
```

In example 7.7 only one copy of the string exists. The functions are given a pointer to this copy and may modify the string. The string is obtained from a call to *scanf* which we have not so far used in the examples on strings. Note that *encode* initialises the array 'code' at its declaration with one digit for each letter of the alphabet. Both *dumpdups* and *dumpzeros* use the expression *t=*(t+1) in a *while* statement to eliminate adjacent identical characters, while *fixup* capitalises upon the flexibility of the *for* statement.

SUMMARY

The availability of arrays has clearly made a significant difference to the ease with which we can express our tasks in C. Pointers, together with arrays, provide us with easy-to-use and economical programming aids. C does not limit us to arrays as a way of storing data items with a collective name. We are also able to use structures, which enable us to group together data items of differing types – this is the subject of the next chaper. Pointers too have a wider role to play than we have thus far indicated, and we will return to them in a later chapter.

The elements of C that we have covered so far constitute a 'basic set'. It is perfectly possible to write meaningful C programs armed with only that knowledge – indeed, the RatC compiler is written using just those features. The remaining chapters deal with more advanced topics, without which your C armoury would be incomplete.

8 More Data Types

So far, all data types of identifiers have been simple: they consist of one elementary type. The elementary types are

(char)	characters
(int)	integers
(float)	floating point

Chars and *ints* can be either signed or unsigned, and *ints* and *floats* can have modifiers short or long. A 'long float' is referred to as a 'double'. Unless otherwise explicitly stated in a declaration, the default type is *int*.

If these were the only data types the C language could represent, many problems would be much more difficult to express than they should be. Part of the great flexibility of C is that the language provides a way to combine elementary types together into new derived types called structures and unions.

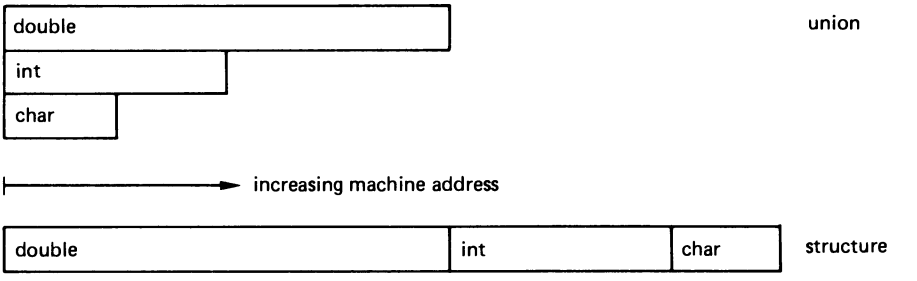
STRUCTURES AND UNIONS

When we combine types, we can do it in one of two ways: we can either lay them end to end so that none of them overlaps and each of them contains independent values, or we can overlay them on top of each other, so that they all start at the same machine storage location and overlap.

If we lay the types next to each other so that none of them overlaps, we create a structure – a type which is the concatenation of the individual member elementary types. Each of the variables starts at a different storage location, one after the other in a series. Therefore, the length of a structure is at least as much as the sum of the lengths of its members. Some compilers insert space in between members of a structure in order to enforce data type address alignment restrictions of the hardware. As a result, the length of a structure may be more than the sum of the lengths of its members because of 'holes' in the structure form.

If we overlay types on top of each other, we create a union – a type which is the union of the individual member elementary types. The same memory storage area is accessed by all of the variables within the union. Since each of the variables starts at the same location, the length of the union is the length of the longest member.

Pictorially, we can represent the distinction between structures and unions as



If we assume that the size of a char is 1 byte, of an int is 2 bytes, and of a double is 4 bytes, then the size of the union is 4 bytes, while the size of the structure is 7 bytes.

Structures are used to group together related data so that it becomes more manageable. Consider, as an example, a date. We can represent the date by three numbers: the month of the year, the day of the month, and the year. By grouping these together, we can create a new type

```
struct date_type {
    short int month;        /* Month of year - 1 --> 12 */
    short int day;          /* Day of month - 1 --> 31 */
    short int year;         /* Year */
};
```

The above statement declares a derived type (*struct* date_type) and its form, that is, what its members are. The identifier date_type is called the structure tag or template name; the compiler will know what a '*struct* date_type' is at any point after this declaration.

No storage is allocated by the above statement, however. The template name before the left curly bracket is used only to identify the form of the structure so that it can be referenced more easily afterwards. To create an instance of this new type to hold a birth date, an identifier is placed after the right curly bracket

```
struct date_type {
    short int month;
    short int day;
    short int year;
} birth;                        /* Date of birth */
```

or better still

```
struct date_type birth;        /* Date of birth */
```

assuming that the template declaration has already been made.

Structures and unions nest; that is, they can be embedded within other structures and unions. Arrays can also be put inside structures or unions. So, if we were interested in storing information about a person, we might create a structure

```
struct person_type {
    char name[NAMESIZE];       /* Name of person */
    struct date_type birth;    /* Date of birth */
    struct date_type death;    /* Date of death */
};
```

We can even create arrays of structures, so that this information about everyone in a group could be stored by declaring

```
struct person_type brits[UK_POPULATION];
```

Unions of all types can be created in a similar fashion. This facility to group data into a new type makes it easier to manage, and thus reduces the complexity of the programming task.

As an example of a union, consider a piece of storage which will sometimes hold an *int*, and at other times a *double*. The declaration for such a union would be written

```
union int_double {
    int i;
    double d;
};
```

ACCESSING STRUCTURES AND UNIONS

Only two things can be done with structures and unions: a member can be accessed, or the address can be taken with the & operator. For the previously declared structure *birth*

```
birth.day
```

represents the member identified by day, and

```
&birth
```

represents the address of that structure. If *pbirth* is declared as a pointer to a date_type structure and then initialised

```
struct date_type *pbirth = &birth;
```

then the day member from such a pointer is accessed with the pointer operator

```
pbirth->day
```

When accessing a member of a structure directly, the dot operator is used; for indirect access from a pointer to a structure, the pointer operator is used.

The name of the person in the first element of the array of structures *brits* declared above is accessed

```
brits[0].name
```

which is an array of characters holding the person's name. Note that this is distinct from the first character of the name, which would be accessed as

```
brits[0].name[0]
```

To give a structure initial values at compile time, it can be declared with them

```
struct person_type henry_viii = {
    "Henry VIII",                /* Name */
    { 6, 28, 1491 },             /* Born June 28, 1491 */
    { 1, 28, 1547 }              /* Died January 28, 1547 */
};
```

Using the dot and pointer operators to access members works with nested structures, so that

```
henry_viii.birth.year
```

would have the value 1491.

Newer compilers understand structure arguments and assignments. That is, structures can be passed as arguments to functions, and are valid as return values from functions. In addition, structure assignment allows the programmer to assign structures of the same type, so that all members are copied in one expression.

ENUMERATIONS

Still another method for creating new types is available with the C language. In an enumerated type, a variable can take on one of a finite set of values which are listed at the place where the type is declared. If we create a type to model the five flavours of ice cream available at a certain store, we could say

```
enum flavour_type {
    CHOCOLATE,
    VANILLA,
    STRAWBERRY,
    COFFEE,
    RASPBERRY
};
```

Thereafter, a variable of type *flavour_type* can take on any of the values enumerated. The values are treated like constants and can be used anywhere constants can be used. Thus

```
enum flavour_type flavour = CHOCOLATE;
```

would create a variable named *flavour*, and give it an initial value of CHOCOLATE.

In our previous example, we could modify the person_type structure to include information about the sex of a person. Since the sex of most people is only one of two possible values, we can define an enumerated type to represent it

```
struct person_type {
    char name[NAMESIZE];        /* Name of person */
    enum sex_type {
        MALE,
        FEMALE
    } sex;                      /* Sex */
    struct date_type birth;     /* Date of birth */
    struct date_type death;     /* Date of death */
};
```

To demonstrate the use of *enum* types, we could write a routine which would recognise an argument of a string of characters as being either 'MALE' or 'FEMALE', and then return the appropriate *enum* value

```
enum sex_type

get_sex (str)

char *str;

{

        return (strcmp (str, "MALE") ? FEMALE : MALE);

}
```

The above routine uses the C library function *strcmp*, which compares two character arrays, and returns an integer which is less than, equal to, or greater than 0 according to whether the first argument is lexicographically less than, equal to, or greater than the second.

BIT FIELDS

There are times when it becomes necessary to pack several pieces of information into the storage that would normally be occupied by a single variable. Such circumstances can occur when manipulating huge amounts of data, or when dealing with boolean values or flags. For these occasions, C provides us with a way to indicate how many bits should be assigned for each variable. When we access one of these fields, the compiler will isolate the correct bits and allow us to manipulate the field as though it was stored as a separate variable. For example, if we wanted to save space and squeeze the date structure so it occupied as little machine storage as possible, we could define it as

```
struct {

    unsigned month : 4;

    unsigned day : 5;

    unsigned year : 11;

} short_date;
```

Since the month of the year can only be a number between 1 and 12, we need only 4 bits to represent it; the day can only be between 1 and 31 (5 bits required), and we can let the year be represented by 11 bits (allows us up to the year 2047). Thus, short_date occupies only 20 bits, instead of the 48 bits it would take if the month, day, and year were each 16 bits (*int*).

There are several restrictions on the use of bit fields – all variables are necessarily unsigned, and there are no arrays of fields. Also, because they might not begin on a byte or word boundary, they have no address, so the & operator cannot be applied to them.

As the cost of memory continues to decline, it seems that bit fields will be most useful in those cases when compact representation of data is paramount.

VOID

An additional type, 'void', is available to describe those objects which have no value. This is useful for declaring functions that return no value, or casting expressions which generate values that are to be discarded. As an example, the function *exit*, which does not return to the calling routine after it is invoked, could be declared

```
void exit ();
```

A void expression denotes a non-existent value and, as such, can only be used as an expression statement, or as the left operand of a comma expression.

TYPEDEF

In C, it is possible to use a short-hand notation to describe fundamental or derived types. A declaration using *typedef* defines synonyms for the indicated type. For example, we could define the data_type structure previously mentioned in this chapter as a *typedef* called DATE in the following manner

```
typedef struct {
    short int month;        /* Month of year - 1 -> 12 */
    short int day;          /* Day of month - 1 -> 31 */
    short int year;         /* Year */
} DATE;
```

After this declaration, the compiler will understand the use of DATE as a reference to the above structure template. It is important to note that no new types are generated; the use of *typedef* is just a short-hand for an existing type. The semantics are exactly the same for *typedef* variables as for variables whose definitions are written out the long way. *Typedefs* can be used to declare synonyms for unions, enums and fundamental data types in exactly the same way.

Arrays, functions and pointers can be used in *typedef* declarations as well. The declaration

```
typedef int ARRAY_DATE[3];
```

allows the definition of a variable

```
ARRAY_DATE date;
```

which is an array of three *ints*. If we wanted to have a synonym for a pointer to a DATE structure, we could write

```
typedef DATE *PDATE;
```

Thus, PDATE would be a pointer to a DATE structure.

SUMMARY

The object of the game in programming is to reduce the complexity of problems to a form where the solution is readily understandable to both the writer and the reader. Derived data types afford us the luxury of defining arbitrarily complex aggregates so that we can group variables together in some logical fashion, where it is sensible to do so. This principle of data abstraction allows us to concentrate on the fundamental ideas of the problem, rather than on the details of its implementation. Without derived data types, it would be impossible to implement the data structures that are required to solve complicated problems. The next chapter deals with the development of these data structures.

9 Pointers Revisited

Our use of pointers so far has been largely restricted to the processing of character strings. In this chapter we will explore much more imaginative uses of this very powerful feature of C. In particular, we will need pointers to simplify the handling of the data structures that are typical of more complex programs.

The data structures used by RatC are of necessity very simple – RatC does not support derived data types (such as structures) which would make the task of the compiler writer so much easier.

Choosing the right data structure to contain the data that the program will manipulate is at least as important as choosing the right algorithm and, in many cases, a poor choice of data structure will lead to a clumsy program.

POINTERS TO STRUCTURES

Given an array of structures of the kind

```
typedef struct {
        int a;
        char b;
        float c;
} STRUCT;

STRUCT array[10];
```

we have two methods of stepping through the array, examining the individual elements. One way we are already familiar with – using subscripts, so that *array[i]* refers to the (i+1)th element (because the first element is subscript 0). The other way is to use a pointer

```
STRUCT *p;
for (i = 0, p = array; i < 10; i++, p++)
    printf ("array[%d] %d %c %f\n", i, p->a, p->b, p->c);
```

Note that when we say 'i++' we mean 'add 1 to i', but when we say 'p++' we mean 'add enough to p to make it point to the next element', and this is precisely what C does. Pointer arithmetic takes account of the underlying type, so that 'p++' means something different if p is a pointer to *char* – in that case, since the underlying type is one byte, p is actually incremented by 1.

It is for this reason that the expressions A[I] and *(A+I) are functionally equivalent, regardless of the type of A.

ALLOCATION OF STORAGE

If we wanted to read lines of text from a file and store them internally for subsequent processing, one way that we could do it is to declare an array of fifty 132-character lines, and read the data into it. The problem with this is that we do not know how many lines there will be, or how long they are. As long as the lines are less than 132 characters, and as long as there are less than 50 lines, then the program will work, even though we may have reserved much more space than we actually need (suppose we only have two 10-character lines!).

We can use space much more economically, and eliminate the restriction on the number of lines we can read in, by allocating space dynamically, as shown in example 9.1.

Example 9.1

```
/* Maximum length of input line */

#define       LINESIZE        132


/* Error handling macro */

#define ERROR(msg)    { fprintf (stderr, "%s\n", msg); exit(1); }


/* Linked list structure */

typedef struct list {
    char text[LINESIZE];
    struct list *next;
} LIST;


LIST *lines = NULL,                 /* Pointer to the head of the list */
     *this_line = NULL,             /* Pointer to the current element */
```

```
    *new_line;                  /* Pointer to a new element */

int eof = 0;                    /* End of file flag */

while (!eof)
    {
      /* Allocate space for a new line */

      if (!(new_line = (LIST *) malloc (sizeof(LIST))))
          ERROR ("Memory exhausted");
/* Initialise next pointer */

new_line->next = (LIST *) NULL;

/* Read in the next line */

if (!gets (new_line->text))
    eof = 1;
else

/* If this is the first line, set head and current pointer to it */

if (!lines)
    lines = this_line = new_line;

/* Otherwise, link current line to new one and advance current line */

else
    this_line = this_line->next = new_line;
}


[ style 78.0 ]
```

Here we have generated a 'linked list' data structure, where each element in the structure, as well as containing the data, has a pointer to the next element in the list. Thus, we finish up with exactly as many elements as there are lines in the input – no more, no less. We could print out the text afterwards by

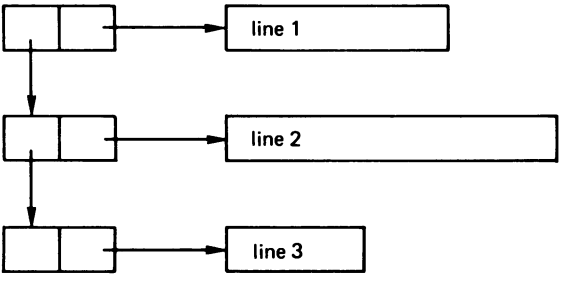
```
for (this_line = lines; this_line; this_line = this_line->next)
    printf ("%s\n" , this_line->text);
```

When allocating space dynamically in this way, it is important to remember that we need to de-allocate, or free, the space at some time. This will be done automatically when the program exits, but if space limitations require that you free the space before then (if, for example, you wish to re-use the space for other purposes), it can be freed by

```
LIST *next_line;

while (lines)
    {
      next_line = lines->next;
      free (lines);
      lines = next_line;
    }
```

and this will leave the variable *lines* set to a NULL value, so that, if used inadvertently, it will not pick up garbage data.

Of course, we have still potentially allocated more space than we need, since each line reserves 132 characters, regardless of its actual length. A better structure would be one that looked like



which would be declared as

```
typedef struct list {
        char *text;
        struct list *next;
} LIST;
```

and we would have to allocate storage for both the list element and for the data, as shown in example 9.2.

Example 9.2

```
/* Maximum line size */

#define BUFSIZE 2048

char data[BUFSIZE];

while (!eof)
    {
      if (!(new_line = (LIST *) malloc (sizeof(LIST))))
          ERROR ("Memory exhausted");

      new_line->next = (LIST *) NULL;
      if (!gets (data))
          eof = 1;
      else
          {

            /* Allocate enough space for this line */

            if (!(new_line->text = (char *) malloc (strlen(data)+1)))
                ERROR ("Memory exhausted");

            /* Copy the line read in */
```

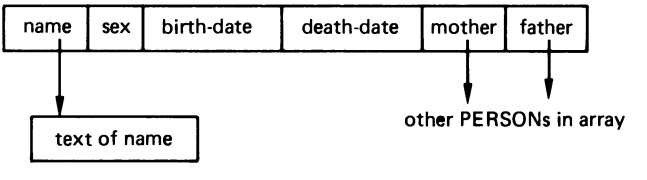
```
        strcpy (new_line->text, data);


        if (!lines)
                lines = this_line = new_line;
            else
                this_line = this_line->next = new_line;
            }
        }


[ style 64.8 ]
```

Now we are allocating exactly the amount of storage required. Note also that the limit on line length is only that it be less than 2048 characters!

COMPLEX DATA STRUCTURES

As an example of a more complex data structure, consider the program of example 9.3, together with its header file in example 9.4. This program constructs a family tree from input data, and prints out the pedigree chart of a named individual.

The principal data structure is an array of elements of type PERSON, which looks like



The dates of birth and death are themselves structures, nested within the PERSON structure.

Example 9.3

```
#include <stdio.h>
#include "family.h"

/* Maximum number of people in input data */

#define MAXPEOPLE      64

/* Error handling macro */
```

```
#define ERROR(msg,data)       \
  { fprintf (stderr,"%s%s\n",msg, data); exit (1); }

/* Array for data structure */

static PERSON people[MAXPEOPLE+1];

/* Pointer to output image area */
static char *space;

/* Months of year */

static char *month[MONTHS] =
    { "JANUARY",   "FEBRUARY",  "MARCH",     "APRIL",
      "MAY",       "JUNE",      "JULY",      "AUGUST",
      "SEPTEMBER", "OCTOBER",   "NOVEMBER",  "DECEMBER" };

/* Global variables */

static int curr_level = 0;
static int max_level = 0;
static int totrows, totcols;

main (argc, argv)
int argc;
char *argv[];
    { char line[LINESIZE];      /* Input line */
      register int i;           /* General purpose counter */
      register PERSON *p;       /* Pointer to data structure */

      /* Arguments can be passed in on the command line as :
                command arg1 arg2 ...
         where argc is the argument count (including the command
         name), and argv[i] are the arguments (argv[0] is the
         command itself, argv[1] is the first argument, etc.) */

      if (argc != 2)
          ERROR ("Usage: ftree <name>", "");

      /* Initialise the data structure */

      for (i = 0; i <= MAXPEOPLE; people[i++].name = NULL)
          ;

/* Input lines consist of fields separated by "tokens"
   from SEPSTRING. Read in each line, extracting the
   fields and entering them into the data structure.
   Ignore lines beginning with "*" (comments). */

while (gets (line))
    {
      if (line[0] == '*')
          continue;
      p = get_name (strtok (line, SEPSTRING));
      p->sex = get_sex (strtok (NULL, SEPSTRING));
      p->birth = get_date (strtok (NULL, SEPSTRING));
      p->death = get_date (strtok (NULL, SEPSTRING));
      p->father = get_name (strtok (NULL, SEPSTRING));
      p->mother = get_name (strtok (NULL, SEPSTRING));
    }

/* Find out how big the tree will be .. */

get_level (p = get_name (argv[1]));
totrows = (5 * power (2, max_level) - 1);
totcols = (max_level + 1) * COLPLEV;
```

```
    /* ... and allocate space for the output */

    if (!(space = malloc ((unsigned) (totrows * totcols))))
        ERROR ("Memory exhausted", "");

    /* Initialise the output area with spaces using the library
       function memset */

    memset (space, (int) ' ', totrows * totcols);

    /* Generate the output image ... */

    drawtree (p, 0, 1);
    vlines ();

    /* ... print it ... */

    printtree ();

    /* ... and exit */

    exit (0);
}

/**
 *  Find the person indicated by the supplied name in the
 *  'people' array. If the person is currently non-existent,
 *  insert them into the array. Return a pointer to the person
 *  if successful, otherwise terminate with an error message.
**/

PERSON *
get_name (str)
register char *str;             /* Name of person */
{ register PERSON *p;           /* Data structure pointer */
  static DATE zero_date = { 0, 0, 0 }; /* Date */

  /* '-' means unknown */

  if (!strcmp (str, "-"))
      return (PERSON * ) 0;

  /* Search the array for a matching name */

  for (p = people; p->name && strcmp (p->name, str); p++)
      ;

  /* If found, return the pointer ... */

  if (p->name)
      return p;

  /* ... otherwise make sure there's enough room ... */

  if (p >= &people[MAXPEOPLE])
      ERROR ("Too many people", "");

  /* ... add them to the end ... */

  if (!(p->name = malloc ((unsigned) strlen (str) + 1)))
      ERROR ("Memory exhausted", "");
  strcpy (p->name, str);
  p->birth = p->death = zero_date;
  p->father = p->mother = (PERSON * ) 0;

  /* ... and return the pointer */
```

```
        return p;
    }

/**
 *  Determine sex.
**/

enum sex_type
get_sex (str)
register char *str;             /* Sex from input line */
    {
      /* Convert to upper case */

      strupcase (str, str);

      /* Should be either MALE or FEMALE */

      return (strcmp (str, "MALE") ? FEMALE : MALE);
    }

/**
 *  Convert src to upper case in dest (toupper is a library
 *  function that converts [a-z] to [A-Z], and leaves all other
 *  characters untouched).
**/

strupcase (dest, src)
register char *dest, *src;   /* Destination and source strings */
    {
      while (*dest++ = toupper (*src++))
          ;
    }
/**
 *  Generate the family tree in the output image by drawing this
 *  person, and then the family trees of their mother and father.
 *  The recursion stops when we run out of parents.
**/

drawtree (p, level, offset)
PERSON *p;
int level, offset;
    { PERSON * mom, *dad;

      /* Draw this person */

      drawperson (p, rowloc (level, offset), level * COLPLEV + 1);

      /* Draw father's family tree */

      for (dad = people; dad->name && dad != p->father; dad++)
          ;
      if (dad->name)
          drawtree (dad, level + 1, offset * 2 - 1);

      /* Draw mother's family tree */

      for (mom = people; mom->name && mom != p->mother; mom++)
          ;
      if (mom->name)
          drawtree (mom, level + 1, offset * 2);
    }

/**
 *  Print date.
**/

 char    *
put_date (date)
DATE date;                      /* Date to be printed */
```

```
    { static char words[25];    /* Buffer for date in words */

      sprintf (words, "%s %d, %d", month[date.month - 1],
        date.day, date.year);
      return words;
    }

/**
 *  Draw a person in the output image, complete with name and dates
 *  of birth and death.
**/

drawperson (p, row, col)
PERSON *p;                      /* Person to be drawn */
int row, col;                   /* Where to draw */
    { char *d;                  /* Date buffer */

      /* Copy in name (memcpy is a library function which copies
         data from its second parameter to its first for a length
         in bytes of its third parameter) ... */

      memcpy (pixel(row, col + 1), p->name, strlen (p->name));
      memcpy (pixel(row + 1, col), NAMELINE,
        sizeof(NAMELINE) - 1);

      /* ... and birth date, if it exists ... */
      if (p->birth.year)
          {
            memcpy (pixel(row + 2, col), " Born", 5);
            d = put_date (p->birth);
            memcpy (pixel(row + 2, col + 6), d, strlen (d));
          }

      /* ... and date of death */

      if (p->death.year)
          {
            memcpy (pixel(row + 3, col), " Died", 5);
            d = put_date (p->death);
            memcpy (pixel(row + 3, col + 6), d, strlen (d));
          }
    }

/**
 *  Print the output image.
**/

printtree ()
    { int i;

      for (i = 0; i < totrows; i++)
          printf ("%.*s\n", totcols, pixel(i + 1, 1));
    }

/**
 *  Put vertical lines into output image.
**/

vlines ()
    { register int i, j, k;

      for (i = 1; i <= max_level; i++)
          for (j = 1; j < power (2, i); j += 2)
              for (k = rowloc (i, j) + 1;
                k <= rowloc (i, j + 1) + 1; k++)
                  *(pixel(k, i * COLPLEV + 1)) = '|';
    }
```

```
/**
 *  Convert str to a date. Terminate with a message on error.
**/

DATE
get_date (str)
register char *str;             /* Date from input line */
    { char *ptr;                /* String pointer */
      register int i;           /* Month counter */
      DATE date;                /* Converted date */

      /* '-' means unknown */

      if (!strcmp (str, "-"))
          date.month = date.day = date.year = 0;

      /* Convert str to DATE format */

      else
          {
            strupcase (str, str);
            for (i = 0; i < MONTHS; i++)
                if (!strncmp (str, month[i], strlen (month[i])))
                    break;
            if (i >= MONTHS)
                ERROR ("Invalid date ", str);
            date.month = i + 1;

            /* strtol is a library function that returns the long
               integer corresponding to the string in the first
               argument according to the number base in the third
               argument.  Leading white space is ignored. If the
               second argument is not NULL, it will contain the
               address of the first non-digit character which
               terminates the conversion. */

            date.day = (short int)
              strtol (str + strlen (month[i]), &ptr, 10);
            date.year = (short int)
              strtol (ptr + 1, (char **) 0, 10);
          }
      return date;
    }

/**
 *  Find out how many generations have to be printed.  This
 *  function operates recursively by determining the number of
 *  generations above this one on both the mother's and father's
 *  side - the number of generations to be printed is the maximum
 *  of these numbers.
**/

get_level (p)
PERSON *p;                      /* Name of person */
    { PERSON * dad, *mom;       /* Pointer to mother & father */

      /* Find father */

      for (dad = people; dad->name && dad != p->father; dad++)
          ;
      /* Find out how many generations above him */

      if (dad->name)
          {
            curr_level++;
```

```
            max_level = max(max_level, curr_level);
            get_level (dad);
            curr_level--;
          }

      /* Find mother */

      for (mom = people; mom->name && mom != p->mother; mom++)
          ;

      /* Find out how many generations above her */

      if (mom->name)
          {
            curr_level++;
            max_level = max(max_level, curr_level);
            get_level (mom);
            curr_level--;
          }
    }

/**
 *  C does not have an exponentiation operator - this function
 *  simulates it.
**/

power (base, exp)
register int base, exp;
    { register int i, result;

      result = 1;
      for (i = 0; i < exp; i++)
          result *= base;
      return result;
    }

/**
 *  Find the row position in the output image for this generation.
**/

rowloc (level, offset)
int level, offset;
    {
      if (level == max_level)
          return (offset * 5 - 4);
      if (level == max_level - 1)
          return (offset * 10 - 6);
      return (rowloc (level + 1, offset * 2) +
              rowloc (level + 1, offset * 2 - 1)) / 2;
    }

      /* ... and birth date, if it exists ... */
      if (p->birth.year)
          {
            memcpy (pixel(row + 2, col), " Born", 5);
            d = put_date (p->birth);
            memcpy (pixel(row + 2, col + 6), d, strlen (d));
          }

      /* ... and date of death */

      if (p->death.year)
          {
            memcpy (pixel(row + 3, col), " Died", 5);
            d = put_date (p->death);
            memcpy (pixel(row + 3, col + 6), d, strlen (d));
```

```
            }
    }

/**
 *  Print the output image.
**/

printtree ()
    { int i;

      for (i = 0; i < totrows; i++)
          printf ("%.*s\n", totcols, pixel(i + 1, 1));
    }

/**
 *  Put vertical lines into output image.
**/

vlines ()
    { register int i, j, k;

      for (i = 1; i <= max_level; i++)
          for (j = 1; j < power (2, i); j += 2)
              for (k = rowloc (i, j) + 1;
                k <= rowloc (i, j + 1) + 1; k++)
                  *(pixel(k, i * COLPLEV + 1)) = '|';
    }


[ style 83.8 ]
```

Example 9.4

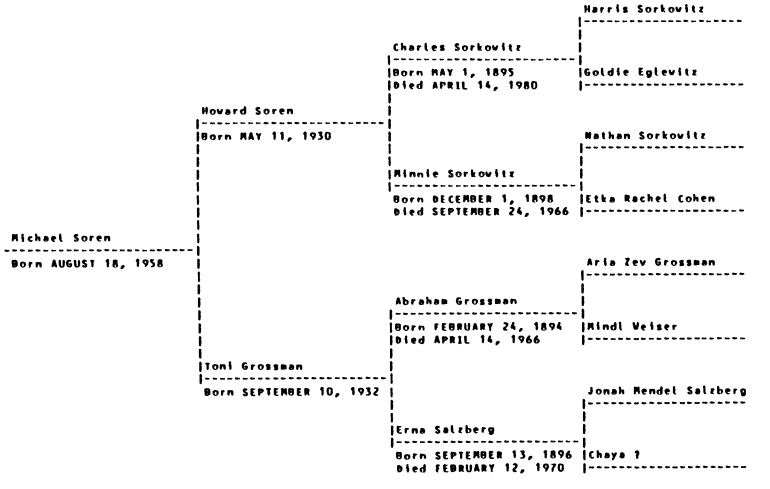
```
typedef struct {
    short int month;            /* Month of year: 1 -> 12   */
    short int day;              /* Day of month: 1 -> 31    */
    short int year;             /* Year: 1 -> 1987          */
} DATE;

typedef struct person {
    char *name;                 /* Name of person           */
    enum sex_type {
        MALE,
        FEMALE
    } sex;                      /* Sex                      */
    DATE birth;                 /* Date of birth            */
    DATE death;                 /* Date of death
                                   (0 year ==> still alive) */
    struct person *mother;      /* Pointer to mother        */
    struct person *father;      /* Pointer to father        */
} PERSON;

/* Maximum length of an input line */
#define LINESIZE      128

/* Valid separators between fields in input line */
#define SEPSTRING     ":\n"

/* Width of one output column */
#define COLPLEV               25

/* Months in a year */
#define MONTHS                12
```

```
/* Maximum width of a name */
#define NAMELINE        "-------------------------"

/* Define NULL if it isn't defined already */
#ifndef NULL
#define NULL            ((char *) 0)
#endif

/* Maximum value of x and y */
#define max(x,y)        ((x) > (y) ? (x) : (y))

/* Position in output array of row r column c */
#define pixel(r,c)      (space + ((r) - 1) * totcols + (c) - 1)

/* Function calls - since all functions not declared are assumed
   to return int, we must declare those that don't */
void exit ();
char *strtok (), *malloc (), *strcpy (), *memset (), *memcpy ();
long strtol ();
PERSON *get_name ();
DATE get_date ();
enum sex_type get_sex ();
```

Input to the program might look like

```
* Input for ftree.c program
* Family tree of Michael Soren
Michael Soren:male:August 18, 1958:-:Howard Soren:Toni Grossman
Toni Grossman:female:September 10, 1932:-:Abraham Grossman:Erna Salzberg
Howard Soren:male:May 11, 1930:-:Charles Sorkowitz:Minnie Sorkowitz
Abraham Grossman:male:February 24, 1894:April 14, 1966:Aria Zev Grossman:Mindl Weiser
Erna Salzberg:female:September 13, 1896:February 12, 1970:Jonah Mendel Salzberg:Chaya ?
Charles Sorkowitz:male:May 1, 1895:April 14, 1980:Harris Sorkowitz:Goldie Eglewitz
Minnie Sorkowitz:female:December 1, 1898:September 24, 1966:Nathan Sorkowitz:Etka Rachel Cohen
```

in which case the output for the pedigree chart of Michael Soren would look like



The program is commented well enough to be self-explanatory, but there are a number of features which are worthy of further explanation. Firstly, there are some 'standard' functions used, such as *memset* and *strtok*, which are part of a run-time library whose contents will depend on the particular installation (see appendix 5). The ones we have used are standard on Unix, but may be different in other implementations. In any case, the functions are mostly straightforward to duplicate.

Secondly, the mechanism for passing arguments into the program from the command line is demonstrated. In order that a program be as flexible as possible, it is important to parameterise it in the same way that you would parameterise any other function. In this case, the parameter is the name of the person whose pedigree chart is to be printed.

Thirdly, notice that the functions *get_level* and *drawtree* are recursive, which is a common feature of programs which manipulate data structures. Any one person's family tree consists of two sub-trees – the family trees of both person's mother and father. *Drawtree* utilises this fact to draw the person's family tree by drawing first the person, and then the family trees of the person's mother and father; *get_level* determines the number of generations to be printed, which is simply one more than the maximum of the number of generations in either the mother's or father's tree.

And finally, note how provision is made for the input data to contain comment lines – this simple feature allows commentary to be included within data files to explain, for example, what the data is, or how it is to be used.

SUMMARY

The theory and practice of data structures is a complicated topic, and one which is largely beyond the scope of this book. What we have presented is the basic tools – pointers, structures and dynamically allocated storage – which will allow you to generate arbitrarily complex data structures.

The thing to remember is that pointers are the equivalent in data structures of gotos in control structures. It is as easy to finish up with unruly data structures as it is to generate 'spaghetti code', and both are usually indicative of lack of forethought. The representation of data requires as much thought as the algorithm which manipulates it, and often the two are inextricably linked, in the sense that a poor design of either may cause the other to be unnecessarily complex and clumsy. The book *Algorithms + Data Structures = Programs* by Wirth (1976) is an excellent illustration of the way in which algorithms and data structures interact.

10 The C Preprocessor

We have already introduced the C preprocessor directives *#include* and *#define* for file inclusion and symbol definition capabilities. In this chapter, we expand the discussion to include the *#undef* directive, and the use of the conditional compilation directives *#if*, *#ifdef*, *#ifndef*, *#else* and *#endif*. In addition, parameters for the *#define* directive are introduced to yield a more powerful macro facility.

Note that the C preprocessor is not part of the compiler; it is a macro processor which is used prior to compilation to perform textual substitutions and file inclusion. It has no knowledge of C syntax, and could equally well be used to process text in any language, including natural language. The results of the processed text are passed to the C compiler for subsequent translation.

#define

#define is used to associate a symbol with a value

```
#define  ENTRIES  100
```

If the value changes, we need only change it in the place where it is declared. A definition may refer to previously defined symbols, as in

```
#define  ARRAYSIZE  (ENTRIES+1)
```

The parentheses surrounding the substitution string are not mere formality; if ARRAYSIZE is used in the following context

```
char array[ARRAYSIZE*4];
```

then omitting the parentheses would allocate an array of 104 bytes (100+1*4) instead of the intended 404 ((100 + 1) * 4).

In chapters 1 and 2, when we discussed the use of the *#define* directive to define constant text, we gave the example

```
#define  CLEAR  printf("\033Y")
```

to define the sequence necessary to clear the screen on a Lear Siegler ADM5.

#undef

To make the preprocessor forget its definition of CLEAR, we can write

```
#undef  CLEAR
```

and thereafter the preprocessor will leave all occurrences of CLEAR alone, passing it unsubstituted to the compiler.

CONDITIONAL COMPILATION

When we write programs, it is advantageous to try to write them in such a way so they are portable; that is, they can be moved to another machine of differing architecture or operating system without changing the source code. They should perform the same function on the new machine as they did on the old one, even though the underlying code and implementation may be different. This increases programmer efficiency so that it is no longer necessary to re-code existing functions for a new machine. The preprocessor makes this task easier with the availability of conditional compilation.

Consider the example of clearing a terminal screen. If all terminals in the world were Lear Siegler ADM5s, the definition of CLEAR would be the same in all cases. However, because different terminals use different sequences to accomplish the same function, this definition must be modified. On a DEC VT100, the statement would have to be

```
#define  CLEAR  printf("\033[2J")
```

The conditional compilation statements allow us to include certain sections of code based upon specified conditions. Thus, we can combine the two CLEAR definitions so that the desired one is defined for either situation. We can write

```
#ifdef  VT100
#define  CLEAR  printf("\033[2J")
#else
#define  CLEAR  printf("\033Y")
#endif
```

The above construction says that if the symbol VT100 is defined to the preprocessor, use the first definition of CLEAR; otherwise, use the second. Conditional compilation proceeds until the *#endif* directive is encountered. Now, all that is needed in order to use this program for a VT100 is to include a line at the top of the program which defines the symbol VT100

```
#define  VT100  1
```

If we wanted to, we could define the sequence for all other available terminals so that the same source code would run unchanged.

We can make similar constructions to define symbols only if they are not already defined, as in the following

```
#ifndef  NULL
#define  NULL  ((char *) 0)
#endif
```

This construction defines the symbol NULL only if it was not previously defined.

We can make the condition for compilation more complex by using the *#if* directive. With the *#if* directive, the condition must be a non-zero constant at compile time in order for the lines through *#endif* to be passed to the compiler. Making programs machine independent then becomes a matter of defining a symbol and testing for it to indicate the target processor. Then, definitions are made on the basis of which type machine the program is compiled for

```
#if mc68k | i286 | i386
    .
    /* Set definitions for the Motorola 68000 based
       or Intel 80286 or 80386 processor */
    .
#endif
#if u3b2 | u3b5 | u3b15 | u3b20
    .
    /* Set definitions for the AT&T 3b processors */
    .
#endif
#if uts | u370
    .
    /* Set definitions for the Amdahl and IBM processors */
    .
#endif
```

Although the examples presented above show only preprocessor directives (*#define*, *#undef*) used within the conditional compilation directives, C source code can be placed there as well to perform different functions under different circumstances.

MACRO PARAMETERS

The *#define* directive is useful in its ability to substitute arbitrary text for a symbol. Here, we see how that capability can be expanded by providing arguments with a macro definition. As an example, consider a macro useful for debugging which prints out a trace message when a function is entered

```
#define  DB_ENTER  printf("Entering a function\n")
```

We could place this statement at the beginning of each function

```
my_function ()
{
    DB_ENTER;
    .
    .
    .
```

This macro, in itself, is not very useful, since it does not say which function is being entered, and the flow of logic may not be easy to understand. Fortunately, we can provide an argument (the function name) with the macro invocation if we define the macro as

```
#define  DB_ENTER(x)  printf("Entering function %s\n", x)
```

Then, the statement at the beginning of each function could look like

```
my_function ()
{
    DB_ENTER("myfunction");
    .
    .
    .
```

After the DB_ENTER macro is substituted, the printf will arrange to print out "Entering function my_function\n", which can be useful in examining the flow of control.

Similarly, we could define a macro to tell us when control is leaving a function, and the returning value. We could define

```
#define  DB_RETURN(x)  {printf("Returning value = %d\n", x); return(x);
```

so that if the above function were written as

```
my_function ()
{
    DB_ENTER("myfunction");
    .
    .
    .
    DB_RETURN(69);
}
```

the output would look like

```
Entering function my_function
Returning value = 69
```

This type of information can be very useful when trying to trace what is happening inside a program.

We could combine this with conditional compilation directives so that output would only be printed if a certain symbol, such as DEBUG, were defined

```
#ifdef DEBUG
#define  DB_ENTER(x)  printf("Entering function %s\n", x)
#define  DB_RETURN(x)  {printf("Returning value = %d\n", x); return(x)
#else
#define  DB_ENTER(x)
#define  DB_RETURN(x)  return(x)
#endif
```

The second definition of DB_ENTER specifies that the DB_ENTER(x) text should be substituted by nothing. Then, the program could be coded as before, but would only produce trace output if it was compiled with the symbol DEBUG defined. If the symbol DEBUG were not defined, no extra code would be generated into the program.

Macro parameters can also be used to simplify complex expressions or structure references. In example 9.4 where a PERSON structure was declared, we could define a macro to access the name of a person's paternal grandfather easily

```
#define GRANDPA(p) (p->father->father.name)
```

SUMMARY

There are many reasons to utilise the C preprocessor's capabilities to perform text substitution within a program. Among them are

- #define'd constants and macros can be declared in one place and used throughout the code; subsequent changes can be made once at the declaration, without having to search for every instance.
- Complexity can be hidden from the programmer without sacrificing efficiency or functionality so that program logic is not obscured by detail.
- Conditional compilation can be used to eliminate machine and other dependencies.
- Using names for constants improves the intelligibility of the code.

11 Programming Style

Programming in any language is a skill acquired largely by experience and by observing the example of others. This is one of the reasons that the compiler listing for RatC, a substantial program, is included as an appendix (the other principal reasons are that, firstly, it is feasible for the interested reader to implement a minimal C system on his own machine, if he does not already have access to the language; and that, secondly, since we have spent the previous chapters discussing the input that is acceptable to the compiler, we thought you might be curious to see the type of C program that processes your C programs).

The way in which your programs are presented is a matter for personal taste. It is often a trade-off between brevity and intelligibility. Although programming 'style' is often considered to be unquantifiable and assessable only in subjective terms, we have made an attempt, in another appendix, to identify those features of program layout and organisation that tend to make it more visually appealing and more easily comprehensible.

It is now realised that the lifetime of a program, and the cost of program maintenance, frequently done by someone other than the author, make considerations of clarity of expression often of equal importance with those of efficiency. It is to the usually conflicting aims of clarity, conciseness and efficiency that we address our attention in this final chapter.

CLARITY

The clarity of a program is influenced by two principal factors: the way in which the program is presented visually, and the way in which the programming language constructs are used. The 'style score' that we have associated with all the programming examples throughout the book is a measure of the former. Appendix 3 gives a suite of programs (not all of which are written in C, although all of them could be) to perform a style analysis on a C program according to certain criteria which we believe contribute directly to a program's readability. You may not agree entirely with the criteria that we have chosen, or with the importance that we attach to each criterion, but you will almost certainly agree that the version of the program 'detab' presented in appendix 3 (which replaces all the tab characters

in a file by the appropriate number of spaces), is very much more intelligible than that presented in example 11.1.

Example 11.1

```
#include        <stdio.h>

main()
{ int c,i,tabs[132],col=1;
  settabs(tabs);
  while ((c=getchar())!=EOF)
      if (c=='\t')
          do {putchar(' ');col++;} while (!tabpos(col,tabs));
      else if (c=='\n') {putchar('\n'); col=1;}
           else {putchar(c); col++;}
}
settabs(tabs)
int tabs[132];
{ int i;
  for (i=1;i<=132;i++)
      if ((i%8)==1) tabs[i]=1;else tabs[i]=0;
}
tabpos(col,tabs)
int col,tabs[132];
{
  if (col>132) return(1);else return(tabs[col]);
}


[ style 24.6 ]
```

The programs are equivalent, in the sense that they contain identical executable statements differently laid out. The 'bad' program could, of course, be very much worse, but then it would not be so typical of the kind of program that it is very tempting to write in a language that encourages brevity. In the authors' experience, programs written like this, with the intention of subsequent cosmetic improvement, tend to remain in their original format – there is little incentive to modify (even superficially) a working program. Automatic aids to 'beautifying' a program by introducing indentation, blank lines, etc. to reflect the program's structure are no substitute for a program thoughtfully written.

The criteria that we have chosen to use in the style analysis of our own programs are shown, in decreasing order of importance, in table 11.1.

Table 11.1

Criterion	Weighting	Ideal range	
Module length	15%	10-25	non-blank lines
Identifier length	14%	5-10	characters
% comment lines	12%	15-25%	
% indentation	12%	24-48%	
% blank lines	11%	15-30%	
Characters per line	9%	12-25	non-blank characters
Spaces per line	8%	4-10	spaces
% #defines	8%	15-25%	of all identifiers
Reserved word usage	6%	16-30	of available words
Include files	5%	3	included files

The relative weights and ideal ranges are not arbitrarily chosen, but rather are the result of careful tuning after analysis of programs that we recognised intuitively as 'good' or 'bad'. They may need modification to cater for individual preferences, or to reflect a particular 'house style'. With the exception of RatC, all examples for which a style score is given are small in size. Style scores for a number of large programs from the UNIX system are given in Berry and Meekings (1985).

The style analysis suite does not pretend to measure, in anything more than the most rudimentary sense, the second factor contributing to clarity: the use of the language itself. As in so many things, in programming there is no 'right' answer – just a number of alternative ways of achieving the same ends. Invariably, some of those ways will be clumsy or obscure. This will most often be the result of either inexperience or poor design – experience of using a language brings with it a number of benefits: for example, being able to 'think in the language' avoids the clumsy type of construct that arises from the direct transliteration of an algorithm derived by a programmer more familiar with another language, and also being able to use effectively the programming 'tricks' that exist within any language (for example, in C, using

```
while (*str1++ = *str2++);
```

to copy a string); and poor initial design, failure to derive a complete solution before coding, is bound to yield a program that is a functional mess, badly structured and with poor lines of communication.

CONCISENESS

There is a point, not always easy to identify, at which 'concise' becomes 'obscure'. Compare, for example, the random number generator program of chapter 6 with the functionally equivalent program of example 11.2. The gain in execution speed would have to be considerable to justify the inclusion of such a complex (but perfectly legal) statement in any program.

Example 11.2

```
#define maxint 32767
#define pshift 4
#define qshift 11

random(range)
    int range;
    {
    static int n=1;

    return((n=((n=n^n>>pshift)^n<<qshift)&maxint)%(range+1));
    }


[ style 49.3 ]
```

As a further example of a program that is concise to the point of obscurity, study the program of example 11.3, and try to determine its effect.

Example 11.3

```
#define LO 2
#define HI 1000

main()
    { int i,j;

      for (i=LO; i<=HI; i++)
          {
            j=sum(i);
            if (j==i) printf("%d\n",i);
            else if (sum(j)==1) printf("%d %d\n",i,j);
          }
    }

sum(n)
```

```
    int n;
    { int s, f;

      s=1;
      for (f=2; f<n; f++)
          if (n%f==0) s+=f;
      return(s);
    }

[ style 49.9 ]
```

Even with explanation, the program is very much more difficult to understand than is the equivalent program of example 11.4 which differs only by using more meaningful identifier names and having a helpful user interface. The program is in fact a generalisation of the perfect number program of chapter 5. Perfect numbers are a special case of 'amicable' numbers, which are pairs of numbers, each of whose sum of factors yields the other number; so that, for example, the sum of the factors of 220 is 284, while the sum of the factors of 284 is 220: 220 and 284 are amicable numbers.

Example 11.4

```
#define LO 2
#define HI 1000

main()
    { int number, sum;

      for (number=LO; number<=HI; number++)
          {
            sum=factorsum(number);
            if (sum==number) printf("%d is perfect\n",number);
            else if (factorsum(sum)==number)
                 printf("%d and %d are amicable\n",number,sum);
          }
    }

factorsum(num)
    int num;
    { int fsum, factor;
```

```
    fsum=1;
    for (factor=2; factor<num; factor++)
        if (num%factor==0) fsum+=factor;
    return(fsum);
}
```

```
[ style 63.4 ]
```

C is undoubtedly a concise language, and encourages the terse representation of complex ideas. Such power should be judiciously used.

EFFICIENCY

The price that is paid for writing programs in any high-level language is in program size and execution time. Unless either of these is particularly critical, the advantages, in terms of productivity and maintenance costs, far outweigh the disadvantages.

C has a number of features that are more usually found in a lower-level language, to the extent that the correspondence between a C program and the machine code to which it compiles is often very close. The effect of this is to reduce the overheads resulting from the translation process very much more than for other contemporary languages. Some C compilers will offer the user an optional optimisation phase, but an alert and informed user is usually the best optimiser of a program. C provides some help in this: for example, the type specifiers *int* or *char* may be preceded by the storage class specifier ***register***, thus

```
register int  n;
register char *sptr;
```

This is interpreted by the compiler as an indication that these identifiers will be heavily used and should, if possible, have storage space in registers. If the compiler is able to do this, then shorter, faster programs should result.

Nevertheless, the program has not yet been written that could not be written better, or executed faster. This was our experience with the RatC compiler. RatC is a descendant, via two generations, of Small-C (Cain, 1980a). For us, Small-C begat HotC, which was a version largely the same structurally as the original, but with considerable modification in the interests of efficiency; and HotC begat RatC which, apart from using a different, two-stage, method of code generation, represented a major restructuring of the program in the interests of style.

The transition from Small-C to HotC was made with the aid of one of the software tools available on the UNIX operating system, under which the programs were developed. This provides the facility of producing an 'execution profile' of a program, in terms of, for each function, the number of times that it was called, and the percentage of total execution time that it accounted for. This

is of obvious benefit, since there is relatively little return from devoting time to improving the efficiency either of functions that are infrequently called, or of those that occupy only a small percentage of the execution time. We were able to concentrate on those areas where our efforts would be most rewarded, with the result of reducing the time taken for the compiler to recompile itself to a quarter of its original value.

As an illustration of the kind of improvements that were made, using the compiler recompiling itself as a yardstick, the top of the 'league table' of the execution profile for the original Small-C was

Function	*Number of calls*	*% of execution time*
alpha	382,521	10.1
findmac	3,594	10.0
astreq	334,421	8.6
numeric	381,794	6.4
an	379,550	5.6

In other words, the three functions *alpha*, *numeric* and *an* (which simply check a character parameter to see whether it is alphabetic, numeric and alphanumeric, respectively) accounted for a quarter of the execution time, and *findmac* (which is essentially a table look-up to determine whether a symbol has been previously defined as a macro) also made a significant contribution. When it is known that the compiler consists of only about 50,000 characters, the number of calls of *alpha*, *numeric* and *an* should cause concern.

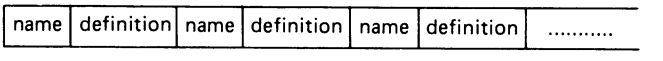
Example 11.5

```
/* test if a given character is alphabetic */
alpha(c)
    char c;
    {
      c=c&127;   /* strip off the parity bit */
      return( ((c>='a') & (c<='z')) |
              ( (c>='A') & (c<='Z')) |
                (c=='_') );
    }

/* test if a given character is numeric */
numeric(c)
    char c;
    {
      c=c&127;
      return( (c>='0') & (c<='9') );
    }
```

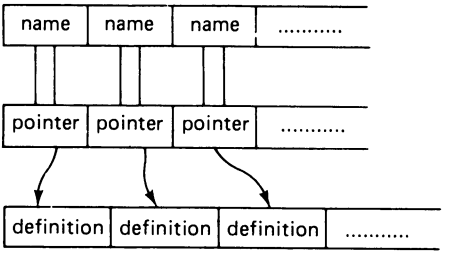
```
/* test if a given character is alphanumeric */
an(c)
    char c;
    {
      return( (alpha(c)) | (numeric(c)) );
    }

[ style 42.0 ]
```

The character checking functions were originally defined as shown in example 11.5. Two significant changes were made: firstly, the parity bit was stripped off once and for all on input by the function *preprocess*, to avoid unnecessary repetition; and secondly, the function *an* was made to check explicitly for the requisite characters, avoiding the overheads incurred by the two function calls. In RatC, these three functions account for less than 5 per cent of the execution time.

The way in which macro definitions were stored was changed from a simple table of the form



to a more complex one of the form



in order to speed up the time taken to perform a linear search for a particular name. This is very important in view of the fact that the majority of searches will be unsuccessful, requiring a search through the entire table. The execution time for *findmac* was thus reduced to a quarter of its original value, at the expense of a little extra memory.

Two points should be mentioned here: firstly, that *findmac* could still be improved, but we believe that the simplicity/efficiency trade-off is about right; and secondly, that the pointer array should strictly be declared as

```
char *macp[mactsize]
```

but, owing to Small-C's inability to handle more than one modifier per declaration, has to be declared (potentially dangerously) as

```
int macp[mactsize]
```

Improving the efficiency of a program is not always an easy, or even desirable, task. For a small program, the effects may not be noticeable; for a large program, run infrequently, the time invested may not be worth while. For a heavily utilised program, such as a compiler, however, attention to the time-critical, bottleneck areas can give a significant improvement in performance.

DEFENSIVE PROGRAMMING

Throughout the book we have attempted to emphasise the importance of the interface between the program and its environment. Any program should take every possible precaution to ensure that it does not fail, and that, if it does, the failure is 'graceful', which is to say that it should provide the naive user with sufficient information to correct, or work around, the problem.

This section is concerned with 'bulletproofing' a program, and consists for the most part of a series of suggestions which you should bear in mind whenever writing programs – they are often the result of painfully acquired experience! If you follow our advice, you are certain to avoid at least some of the common pitfalls of porting programs from one machine to another, which, contrary to popular opinion, is not nearly as simple as it is supposed to be.

(a) Use lint. 'Lint' is a Unix utility which is commonly available on a variety of other systems. It performs a much more rigorous check than does the compiler on such things as type consistency, use of uninitialised variables, and correspondence between actual and dummy function arguments. If we had only one piece of advice to give you, it would be this.

(b) Check input data. At the end of chapter 3, we mentioned that input data is nearly always beyond the control of the programmer. You should check the integrity of all data which is derived from outside the program to make sure that it is within prescribed values. If you do not know what the prescribed values are, at least check that the value will not cause a run-time error – zero values used for division are an obvious example.

(c) Check function arguments. By a similar reasoning to the previous point, if you assume that function arguments are always sane, you will be caught unawares when, at some time in the future, you 'steal' the code to put in some other program where you have not been quite so careful.

(d) Check return values from functions. If a function (either yours or a system-provided one) returns a value, check it before continuing. Nearly all system-provided functions return values, and it is good practice to make yours do so too. Never assume that a function will always be successful – it always will be, except when you do not check it!

(e) Make sure variables are unique. Some C implementations only discriminate

identifiers based on the first 8 characters. If all your identifiers are unique over the first 8 characters, you should have no problem porting your code to a different machine.

(f) Do not rely on uninitialised variables. Variables of storage class *static* can be safely assumed to start with zero value; variables of storage class *automatic* start with garbage values. While this may be true, if you do not explicitly initialise them, the time will come when you change the storage class of one of your variables without changing the program logic, and wonder why it does not work anymore.

(g) Do not exploit implementation-dependent features. On some systems, a pointer occupies the same storage space as an integer. If you use that fact, your program probably will not work on another, dissimilar, machine. Structure-to-structure assignment is available on many systems, but is not yet standard, and should be avoided.

A slightly more insidious example arises from something we said at the end of chapter 6 – 'no ordering is implied among operators with the same priority'. Parentheses in an expression control precedence and associativity, but not order of evaluation, which is to say that the expression '$a + b + c$' could be evaluated by adding a to b, and adding the result to c, or by adding b to c, and the result to a. Normally this causes no problem, but consider the expression

```
y = x++ + x;
```

If x initially has the value 1, what value does y have after the assignment? 2? 4? The answer is that it is impossible to say – of course, on any particular implementation, it will always be evaluated the same way, but this is not true of the same program running on a different machine.

The assignment should have been written as

```
x++; y = x + x;
```

or

```
y = x + x; x++;
```

depending on what you intend.

(h) Do not use side effects in macro calls. The seemingly innocuous macro

```
#define MAX(a,b)  (a < b ? b : a)
```

when invoked by

```
z = MAX (x++, y);
```

leaves x with a different result depending on whether it is greater or less than y, because the preprocessor only performs textual substitution so that, in practice, the macro expands to

```
z = (x++ < y ? y : x++);
```

(i) Use parentheses in expressions. If you are unsure of operator precedence, or if the expression you are formulating is complex, do not be afraid to use parentheses to make it clearer. It adds nothing to the execution time, but a great deal to the comprehensibility.

(j) Do not corrupt C with the preprocessor. It is very easy, using the preprocessor, to make C look like some other language. If you are fond of Pascal, you might be tempted to write

```
#define BEGIN    {
#define END      }
        .
        .
        .
```

but the result will be a confusion of neither one language nor the other.

(k) Use the right type of variable. Do not use an *int* when a *char* will do – for example, with a truth value; or an *int* where you mean a pointer. You not only save space, you give a program like 'lint' a much better chance of detecting potential problems.

(l) Exit gracefully. A program should never fail inexplicably – provide the user with sufficient information as to the cause of the failure that he understands what has gone wrong and what he can do to correct it.

(m) Do not rely on defaults. Often a system-provided function will offer default values for some of its arguments. If you take advantage of that you run the risk of your program no longer working should those defaults ever change.

(n) And lastly, beware of the difference between = and = =. If you are used to a language which uses the same operator for assignment and equivalence, sometime you will fall into the trap. It sounds easy to remember, but we have all forgotten it!

SUMMARY

Programming style and program efficiency are contentious issues: some will maintain that 'style' is so personal that it is impossible to lay down more than vague guidelines, others that it is the business of compilers and optimisers to worry about efficiency. What should never be forgotten is that, as we said in the introduction,

programming is communication, and the communication operates at different levels: between the program and the computer, between the program and the user, and between the program and its maintainer.

It is all too tempting in a language like C to sacrifice clarity for conciseness and efficiency. There are relatively few occasions on which careful consideration of the method by which a program achieves its results (as in the RatC macro table organisation, above) would not yield the desired effect, without the need to resort to tricky obscure code.

The power of C, used properly, can be exploited to produce programs that are elegant, concise and, above all, intelligible.

Appendix 1: RatC

RatC ('rationalised Small-C') is a descendant, via two generations, of Ron Cain's Small-C (Cain, 1980a). Small-C is a compiler for a subset of the full C language, designed originally to generate code for Intel's 8080 microprocessor.

There are obvious reasons why it is advantageous to be able to develop microprocessor programs in a high-level language rather than assembly language, but, in the compiler, this objective inevitably leads to a trade-off between the language features implemented and the compiler size. While Small-C implements only those features of C considered essential to microprocessor applications, the language is rich enough to enable the compiler to recompile itself, so that it is quite possible to contemplate extending the compiler to accommodate any missing features. Modifications and extensions to the original Small-C can be found in Hendrix (1982).

There are three principal reasons for the inclusion of the RatC compiler listing within this book: firstly, because it has been a useful source of programming examples throughout the book; secondly, because any new programming language, in our experience, is most effectively learnt by example rather than by instruction, and the compiler is a good example of a substantial program; and, thirdly, because it is a useful piece of software in its own right.

THE DEVELOPMENT OF RatC

The language actually processed by RatC is identical to that of Small-C: the difference is one of approach and implementation, rather than of end product. Where Small-C generates code suitable for input to an 8080 assembler, RatC generates an intermediate language of its own, closely modelled on a subset of the 8080 instruction set. The resultant code can be executed in one of three ways: it can be interpreted, by a program resident on the target machine; it can be microcoded, effectively changing the target machine's native instruction set (the interested reader is referred to Hall (1982) for an example of a microcoded architecture for a Pascal system); or it can be translated into the specific code for a 'target' machine. We have chosen the last alternative in our implementations.

Whichever method is chosen, the compiler is no longer processor-specific, and it is usually a straightforward task to implement RatC on any particular processor.

THE HYPOTHETICAL MACHINE

The underlying hypothetical machine on which the intermediate instruction set is based is 8080-like, consisting of four registers:

PC	the program counter;
SP	the stack pointer;
P	the primary register; and
S	the secondary register.

The stack, as in most hardware implementations on the older microprocessors, grows downwards. The primary and secondary registers provide a useful extension to purely stack-oriented hypothetical machines: acknowledging the existence of working registers reduces the number of memory accesses required to emulate the machine, while more than two such registers introduce problems of optimal register allocation.

THE INSTRUCTION SET

The instruction set is designed to be translatable to any target instruction set using simple string processing techniques, such as are found, for example, in a text editor or macroprocessor. This will generally produce assembly language to be processed by the native assembler to resolve symbolic addresses. The instruction set is defined as follows

start		beginning of program – perform processor-specific initialisation and housekeeping
ldir.b	<label>	load the byte at the specified address into the low-order byte of the primary register (P), sign extending to the left
ldir.w	<label>	load the word at the specified address into P
addr	<value>	get effective address (SP + offset) into P
sdir.b	<label>	store the low-order byte from P at the specified address
sdir.w	<label>	store P at the specified address
sind.b		store the low-order byte from P at the address in the secondary register (S)
sind.w		store P at the address in S
lind.b		load byte at address in P into P, sign extending to the left
lind.w		load word at address in P into P
call	<label>	call specified subroutine, return address to stack top (T)
scall		call subroutine at address in T, return address to T
ujump	<label>	unconditional jump to specified location
fjump	<label>	jump to specified location if P is false (zero)
modstk	<value>	modify SP by the specified amount
swap		exchange contents of P and S

limm	<expr>	load the specified value into P
push		push P on to stack
pop		pop top of stack into S
xchange		exchange contents of P and T
return		return from subroutine: address on top of stack
scale	<value>	multiply P by the specified value (for example, to convert an integer offset into a byte offset)
add		add S to P, result in P
sub		subtract P from S, result in P
mult		multiply S by P, result in P
div		divide S by P, quotient in P, remainder in S
mod		divide S by P, remainder in P, quotient in S
or		bitwise inclusive or of P and S, result in P
xor		bitwise exclusive or of P and S, result in P
and		bitwise and of P and S, result in P
asr		arithmetic shift right of S, number of places in P, result in P
asl		arithmetic shift left of S, number of places in P, result in P
neg		two's complement of P, result in P
inc	<value>	increment P by the specified value
dec	<value>	decrement P by the specified value
testeq		set P to one if S = P, otherwise reset P to zero
testne		set P to one if S ! = P, otherwise reset P to zero
testlt		set P to one if S < P, otherwise reset P to zero
testle		set P to one if S < = P, otherwise reset P to zero
testgt		set P to one if S > P, otherwise reset P to zero
testge		set P to one if S > = P, otherwise reset P to zero
testult		set P to one if S < P (unsigned), otherwise reset P to zero
testule		set P to one if S < = P (unsigned), otherwise reset P to zero
testugt		set P to one if S > P (unsigned), otherwise reset P to zero
testuge		set P to one if S > = P (unsigned), otherwise reset P to zero
ds	<value>	reserve specified number of bytes of storage
db	<b1,b2,..>	initialise successive bytes of storage as specified
end		end of program

In the above, <label> is an alphanumeric symbolic location; <value> is a decimal, possibly signed, constant; <expr> is either a <value> or an expression of the form <label> + <value>; and <bi> are decimal byte values.

EXAMPLE PROGRAM

The output of the RatC compiler is illustrated overleaf, using a small program that employs a recursive function to print a number in a specified number base (shown in

example A1.1). Source language statements appear as comments in the RatC output and are immediately followed by the code that represents them.

Example A1.1

```
int test, base;

main()              /*  RatC example program   */
    {
      test = 99;
      base = 8;
      prnum(test,base);
      putchar('\n');
    }

prnum(num, base) /* print `num' in base `base' */

    int num, base;
    { int quot,rem;

      if (base<2 | base>10) return;
      if (num<0) { putchar('-'); num= -num; }
      quot=num/base;
      rem=num%base;
      if (quot!=0) prnum(quot,base);
      putchar(rem+'0');
    }

[ style 48.0 ]
```

RatC COMPILER OUTPUT

```
; Lancaster RatC compiler                    modstk  2
;int test, base;                             addr    8
;main()              /*  RatC exampl        push
        start                                addr    10
_main:                                       lind.w
                                             neg
;    {                                       pop
;      test = 99;                            sind.w
        limm    99
        sdir.w  test                ;      quot=num/base;
                                    cc4:
```

```
;       base = 8;
        limm    8
        sdir.w  base

;       prnum(test,base);
        ldir.w  test
        push
        ldir.w  base
        push
        call    _prnum
        modstk  4

;       putchar('\n');
        limm    10
        push
        call    _putchar
        modstk  2

;   }
        return

;prnum(num, base) /* print `num' in
_prnum:

;   int num, base;
;   { int quot,rem;

modstk  -2                  ;       if (quot!=0) prnum(quot,base);
modstk  -2                          addr    2
                                    lind.w
if (base<2 | base>10) return;       push
addr    6                           limm    0
lind.w                              pop
push                                testne
limm    2                           fjump   cc5
pop                                 addr    2
testlt                              lind.w
push                                push
                                    addr    2
                                    push
                                    addr    10
                                    lind.w
                                    push
                                    addr    10
                                    lind.w
                                    pop
                                    div
                                    pop
                                    sind.w

;       rem=num%base;
                                    addr    0
                                    push
                                    addr    10
                                    lind.w
                                    push
                                    addr    10
                                    lind.w
                                    pop
                                    div
                                    swap
                                    pop
                                    sind.w
```

```
        addr    8
        lind.w
        push
        limm    10
        pop
        testgt
        pop
        or
        fjump   cc3
        modstk  4
        return

;       if (num<0) { putchar('-'); nu
cc3:
        addr    8
        lind.w
        push
        limm    0
        pop
        testlt
        fjump   cc4
        limm    45
        push
        call    _putchar
        addr    8
        lind.w
        push
        call    _prnum
        modstk  4

;       putchar(rem+'0');
cc5:
        addr    0
        lind.w
        push
        limm    48
        pop
        add
        push
        call    _putchar
        modstk  2

;       }
        modstk  4
        return
test:   ds 2
base:   ds 2
        end     start
```

RatC *VERSUS* C

The RatC compiler, being written in precisely that subset of the C language that it processes, is the definitive specification of the subset. Briefly, the features that are *not* supported are

(1) Data types other than character, integer and pointer (no floating point).
(2) Structures, unions and multiple-dimension arrays (single-dimension arrays only).
(3) Type definitions.
(4) The logical operators && and | | (bitwise operators are used instead).
(5) The unary operators ! and ~.
(6) The , operator.
(7) Assignment operators other than = (+=, −=, /=, etc.).
(8) Switch, do-while, for and goto statements.
(9) Input/output (other than via your own runtime library).

The preprocessor directives #include and #define are supported, except that *include* files may not themselves contain #include directives, and constant definitions may not be parameterised. An additional directive pair, #asm-#endasm, is provided, between which native assembly language for the target machine can be inserted (use of this feature, of course, makes a program highly machine-dependent).

RatC IMPLEMENTATION

The RatC hypothetical machine and intermediate language are intended to be easily implementable on any target machine. Inevitably, efficiency is sacrificed for generality, and translation from the intermediate language to the target language will not usually yield optimal code.

What this means in practical terms is that, firstly, it will almost always be possible to produce a more compact and faster program than that produced by RatC either by writing directly in native machine code, or by using a compiler that generates code specifically for the target machine, and can thus exploit features of its instruction set that are unknown to RatC; and, secondly, that there will rarely be a one-to-one correspondence between the RatC instruction set and the native instruction set. The more powerful RatC instructions will often be implemented as calls to a runtime library to avoid the excessive space overheads of including the code inline (the same argument applies here as was presented in chapter 2 in support of the choice between macros and functions).

Thus, for any implementation, a runtime library must be provided, whose extent will depend largely on the power of the instruction set for the particular processor. A part of this library that will be common to all implementations, however, is that dealing with input and output. The routines used by the RatC compiler, which are closely modelled on the C standard I/O package, are

fopen	open a file for reading ("r") or writing ("w"), returning a non-zero integer for success, and NULL (0) for failure
fclose	close a file
getc	read in a character from the specified file
getchar	read in a character from the standard input
gets	read in a character string from the standard input
putc	write a character to the specified file
putchar	write a character to the standard output
puts	write a character string to the specified file

As examples of implementation, presented overleaf are the translations from RatC to, at one end of the scale, the 8080 instruction set, and, at the other, to the VAX. We are greatly indebted to Peter Hurley for the VAX implementation.

EXAMPLE TRANSLATION: RatC to 8080

To implement RatC on the 8080, the implementation-dependent constants 'intwidth' and 'charwidth' at the beginning of the compiler need to be changed to 2 and 1 respectively, and a runtime library must be provided, comprising the following routines (in which the primary register is HL, and the secondary DE)

ccgchar fetch a single byte from the address in HL and sign extend into HL
ccgint fetch a 16-bit integer from the address in HL into HL
ccpchar store a single byte from HL at the address in DE
ccpint store a 16-bit integer in HL at the address in DE
ccor, ccxor, ccand
inclusive or, exclusive or, and HL and DE into HL
cceq, ccne, ccgt, ccle, ccge, cclt
set HL to 1 if DE==HL, DE!=HL, DE>HL, DE<=HL, DE>=HL, DE<HL, and to 0 otherwise
ccuge, ccult, ccugt, ccule
set HL to 1 if DE>=HL, DE<HL, DE>HL, DE<=HL (all unsigned comparisons), and to 0 otherwise
ccasr, ccasl
shift DE arithmetically right, left by number of places in HL, and return result in HL
ccsub subtract HL from DE, and return result in HL
ccneg form the two's complement of HL
ccmult multiply DE by HL, and return result in HL
ccdiv divide DE by HL, and return quotient in HL, remainder in DE

The translation below is largely extracted from the original Small-C compiler (Cain, 1980a) the code for the runtime library appears in Cain (1980b).

```
ldir.b  <label>     LDA   <label>
ldir.w  <label>     LHLD  <label>
addr    <value>     LXI   H,<value>
                    DAD   SP
sdir.b  <label>     MOV   A,L
                    STA   <label>
sdir.w  <label>     SHLD  <label>
sind.b              CALL  ccpchar
sind.w              CALL  ccpint
lind.b              CALL  ccgchar
lind.w              CALL  ccgint
call    <label>     CALL  <label>
scall               LXI   H,S+5
                    XTHL
                    PCHL
```

ujump	<label>	JMP	<label>
fjump	<label>	MOV	A, H
		ORA	L
		JZ	<label>
modstk	<value>	XCHG	
		LXI	H, <value>
		DAD	SP
		SPHL	
		XCHG	
swap		XCHG	
limm	<expr>	LXI	H, <expr>
push		PUSH	H
pop		POP	D
xchange		XTHL	
return		RET	
scale	<value>	XCHG	
		PUSH	H
		LXI	H, <value>
		CALL	ccmult
		POP	D
add		DAD	D
sub		CALL	ccsub
mult		CALL	ccmult
div		CALL	ccdiv
mod		CALL	ccdiv
		XCHG	
or		CALL	ccor
xor		CALL	ccxor
and		CALL	ccand
asr		CALL	ccasr
asl		CALL	ccasl
neg		CALL	ccneg
inc	<value>	XCHG	
		PUSH	H
		LXI	H, <value>
		DAD	D
		POP	D
dec	<value>	XCHG	
		PUSH	H
		LXI	H, −<value>
		DAD	D
		POP	D
testeq		CALL	cceq
testne		CALL	ccne

```
testlt                  CALL    cclt
testle                  CALL    ccle
testgt                  CALL    ccgt
testge                  CALL    ccge
testult                 CALL    ccult
testule                 CALL    ccule
testugt                 CALL    ccugt
testuge                 CALL    ccuge
ds      <value>         DS      <value>
db      <b1,b2,..>      DB      <b1,b2,..>
end                     END
```

EXAMPLE TRANSLATION: RatC to VAX

The VAX being a 32-bit word machine, the implementation-dependent constant 'intwidth' in the RatC compiler needs to have the value 4; the only runtime library needed comprises the input/output routines.

```
start                   .ENTRY  START,^M<>
                        CALLS   #0,INIT_IO
                        JSB     MAIN
                        $EXIT_S

ldir.b  <label>         CVTBL   <label>,R2

ldir.w  <label>         MOVL    <label>,R2

addr    <offset>        ADDL3   #<offset>,SP,R2

sdir.b  <label>         CVTLB   R2,<label>

sdir.w  <label>         MOVL    R2,<label>

sind.b                  MOVB    R2,(R3)

sind.w                  MOVL    R2,(R3)

lind.b                  CVTBL   (R2),R2

lind.w                  MOVL    (R2),R2

call    <label>         JSB     <label>

scall                   JSB     (SP)
```

```
ujump   <label>         JMP     <label>

fjump   <label>         TSTL    R2
                        BNEQ    .+4
                        BRW     <label>
modstk  <value>         ADDL    #<value>,SP

swap                    PUSHL   R2
                        MOVL    R3,R2
                        POPL    R3

limm    <value>         MOVL    #<value>,R2

push                    PUSHL   R2

pop                     POPL    R3

xchange                 POPL    R4
                        PUSHL   R2
                        MOVL    R4,R3

return                  RSB

scale   <value>         MULL    #<value>,R2

add                     ADDL    R3,R2

sub                     SUBL3   R2,R3,R2

mult                    MULL    R3,R2

div                     MOVL    R3,R4
                        BLSS    .+5
                        CLRL    R5
                        BRB     .+8
                        MOVL    #-1,R5
                        EDIV    R2,R4,R2,R3

mod                     MOVL    R3,R4
                        BLSS    .+5
                        CLRL    R5
                        BRB     .+8
                        MOVL    #-1,R5
                        EDIV    R2,R4,R3,R2
```

```
or                      BISL    R3,R2

xor                     XORL    R3,R2

and                     MCOML   R3,R3
                        BICL    R3,R2
                        MCOML   R3,R3

asl                     ASHL    R2,R3,R2

asr                     MNEGL   R2,R2
                        ASHL    R2,R3,R2

neg                     MNEGL   R2,R2

inc     <value>         ADDL    #<value>,R2

dec     <value>         SUBL    #<value>,R2

testeq                  CMPL    R3,R2
                        BEQL    .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

testne                  CMPL    R3,R2
                        BEQL    .+6
                        MOVL    #1,R2
                        BRB     .+3
                        CLRL    R2

testlt                  CMPL    R3,R2
                        BLSS    .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

testle                  CMPL    R3,R2
                        BLEQ    .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2
```

```
testgt                  CMPL    R3,R2
                        BGTR    .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

testge                  CMPL    R3,R2
                        BGEQ    .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

testult                 CMPL    R3,R2
                        BLSSU   .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

testule                 CMPL    R3,R2
                        BLEQU   .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

testugt                 CMPL    R3,R2
                        BGTRU   .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

testuge                 CMPL    R3,R2
                        BGEQU   .+5
                        CLRL    R2
                        BRB     .+4
                        MOVL    #1,R2

ds      <value>         .BLKB   <value>

db      <b1,b2,..>      .BYTE   <b1,b2,..>

end                     .END
```

Appendix 2: RatC Listing

```
 1     /****************************************/
 2     /*                                      */
 3     /*               RatC                   */
 4     /*                                      */
 5     /*      Lancaster implementation        */
 6     /*                by                    */
 7     /*    Bob Berry and Brian Meekings      */
 8     /*             May 1983                 */
 9     /*                                      */
10     /*   (based on Ron Cain's  Small-C)     */
11     /*                                      */
12     /****************************************/
13
14
15     /* Implementation dependent definitions */
16
17 #include    <stdio.h>       /* UNIX i/o     */
18 #define     intwidth    2   /* integer size */
19 #define     charwidth   1   /* char size    */
20 #define     clearscreen {putchar(30);putchar(27);putchar('Y');}
21
22     /* Ascii codes */
23
24 #define     bspch       8
25 #define     tabch       9
26 #define     eolch       10
27 #define     ffch        12
28 #define     crch        13
29 #define     quoch       39
30 #define     bslch       92
31
32
33     /* Output definitions */
34
35 #define     nl          outbyte(eolch)
36 #define     tab         outbyte(tabch)
37 #define     colon       outbyte(':')
38 #define     comma       outbyte(',')
39 #define     space       outbyte(' ')
40 #define     comment     outbyte(';')
41
42
43     /* Define the symbol table parameters */
44
```

```
45 #define     symsiz      15
46 #define     symtbsz     5400
47 #define     numglbs     300
48 #define     startglb    symtab
49 #define     endglb      startglb+numglbs*symsiz
50 #define     startloc    endglb+symsiz
51 #define     endloc      symtab+symtbsz-symsiz
52
53
54      /* Define symbol table entry format */
55
56 #define     name        0
57 #define     ident       10
58 #define     type        11
59 #define     storage     12
60 #define     offset      13
61
62      /* System wide name size (for symbols) */
63
64 #define     namesize    10
65 #define     namemax     9
66
67
68      /* Define possible entries for "ident" */
69
70 #define     variable    1
71 #define     array       2
72 #define     pointer     3
73 #define     function    4
74
75
76      /* Define possible entries for "type" */
77
78 #define     cchar       1
79 #define     cint        2
80
81
82      /* Define possible entries for "storage" */
83
84 #define     statik      1
85 #define     stkloc      2
86
87
88      /* Define the "while" statement queue */
89
90 #define     wqtabsz     100
91 #define     wqsiz       4
92 #define     wqmax       wq+wqtabsz-wqsiz
93
94
95      /* Define entry offsets in while queue */
96
97 #define     wqsym       0
98 #define     wqsp        1
99 #define     wqloop      2
```

```
100 #define     wqlab       3
101
102
103     /* Define the literal pool */
104
105 #define     litabsz     2500
106 #define     litmax      litabsz-1
107
108
109     /* Define the input line */
110
111 #define     linesize    80
112 #define     linemax     linesize-1
113
114
115     /* Define the macro (define) pool */
116
117 #define     macbsize    1500
118 #define     mactsize    75
119 #define     macssize    750  /*namesize*mactsize*/
120
121
122     /* Define statement types (tokens) */
123
124 #define     stif        1
125 #define     stwhile     2
126 #define     streturn    3
127 #define     stbreak     4
128 #define     stcont      5
129 #define     stasm       6
130 #define     stexp       7
131
132
133     /* Now reserve some storage words */
134
135 char    symtab[symtbsz];    /* symbol table         */
136 char    *glbptr,*locptr;    /* ptrs to next entries */
137 int     wq[wqtabsz];        /* while queue          */
138 int     *wqptr;             /* ptr to next entry    */
139
140 char    litq[litabsz];      /* literal pool         */
141 int     litptr;             /* ptr to next entry    */
142
143 char    macb[macbsize];     /* macro body buffer    */
144 char    mact[macssize];     /* macro name table     */
145 char    *macbptr  *mactptr; /* and their indices    */
146 char    *macbmax;           /* end of body buffer   */
147 int     macp[mactsize];     /* ptrs into body       */
148 int     macpmax, *macpptr;  /* macro count          */
149
150 char    line[linesize];     /* parsing buffer       */
151 char    mline[linesize];    /* temp macro buffer    */
152 char    *ch,*nexch,*chmax,*mptr; /* ptrs into each  */
153 int     mpmax;              /* limit of mline       */
154
```

```
155
156     /* Miscellaneous storage */
157
158 int     nxtlab,     /* next available label #       */
159         litlab,     /* label # assigned to lit pool */
160         sp,         /* stack pointer                */
161         argstk,     /* function argument sp         */
162         ncmp,       /* # open compound statements   */
163         errcnt,     /* # errors in compilation      */
164         inscnt,     /* # instructions generated     */
165         lncnt,      /* # source lines               */
166         eof,        /* non-zero on final input eof  */
167         input,      /* iob # for input file         */
168         output,     /* iob # for output file        */
169         input2,     /* iob # for "include" file     */
170         glbflag,    /* non-zero if internal globals */
171         mulfile,    /* non-zero for many input files*/
172         ctext,      /* non-zero to intermix c-source*/
173         cmode,      /* non-zero while parsing c-code*/
174                     /* zero when parsing assembly   */
175         laststst;   /* last executed statement type */
176
177 char    quote[2];   /* literal string for '"'       */
178 char    *cptr;      /* work ptr to any char buffer  */
179 int     hasmain;    /* does this file have `main()' */
180
181
182
183     /*****************************************/
184     /*                                       */
185     /*    Compiler begins execution here     */
186     /*                                       */
187     /*****************************************/
188
189 main()
190     {
191     glbptr=startglb;        /* clear global symbols */
192     locptr=startloc;        /* clear local symbols  */
193     chmax =line+linemax;    /* max value of ptr ch  */
194     mpmax =mline+linemax;   /* ditto for mline      */
195     wqptr=wq;               /* clear while queue    */
196
197     litptr=                 /* clear literal pool   */
198     sp=                     /* stack ptr (relative) */
199     eof=                    /* not eof yet          */
200     input=                  /* no input file        */
201     input2=                 /* or include file      */
202     output=                 /* no open units        */
203     ncmp=                   /* no open compounds    */
204     laststst=               /* no last stmt yet     */
205     0;                      /* ... all set to zero  */
206
207     errcnt=                 /* no errors            */
208     inscnt=                 /* no instructions yet  */
209     lncnt=                  /* no source lines yet  */
```

```
210     hasmain=                /* no main segment yet  */
211     0;                      /* ... all set to zero  */
212
213     mactptr=mact;           /* clear the macro table*/
214     macbptr=macb;           /* start of body buffer */
215     macpptr=macp;           /* none to start with   */
216     macpmax=macp+(mactsize-1)*intwidth;
217     macbmax=macb+macbsize-1;
218
219     quote[0]='"';           /* fake a quote literal */
220     quote[1]=0;
221     cmode=1;                /* enable preprocessing */
222
223     /************************/
224     /*    Compiler body     */
225     /************************/
226
227     ask();                  /* get user options     */
228     openout();              /* get an output file   */
229     openin();               /* & initial input file */
230     comment;
231     outstr(" Lancaster RatC compiler");
232     nl;
233     parse();                /* process ALL input    */
234     dumplits();             /* dump literal pool    */
235     dumpglbs();             /* and all static memory*/
236     trailer();              /* follow-up code       */
237     closeout();             /* close the output     */
238     errorsummary();         /* summarize errors     */
239     return;                 /* then exit to system  */
240     }
241
242     /*********************************/
243     /*    Process all input text     */
244     /*  At this level, only static   */
245     /*    declarations, defines,     */
246     /*    includes and function      */
247     /*    definitions are legal      */
248     /*********************************/
249
250 parse()
251     {
252     while (eof==0)          /* do until no more input */
253         {
254         if (amatch("char",4))
255             {declglb(cchar);needsemi();}
256         else if (amatch("int",3))
257             {declglb(cint);needsemi();}
258         else if (match("#asm"))
259             doasm();
260         else if (match("#include"))
261             openinclude();
262         else if (match("#define"))
263             addmac();
264         else newfunc();
```

```
265         skipblanks();     /* force eof if pending   */
266         }
267     }
268
269     /*********************************/
270     /*   New function definition     */
271     /*********************************/
272
273 newfunc()
274     {
275     char n[namesize], *ptr;
276     int  argtop;
277
278     if (symname(n)==0)
279         {
280         error("illegal function or declaration");
281         resetptr();
282         return;
283         }
284     if (ptr=findglb(n))
285         {
286         if (ptr[ident]!=function) errmulti(n);
287         else if (ptr[offset]==function) errmulti(n);
288         else ptr[offset]=function;
289         }
290     else addglb(n,function,cint,function);
291
292     if (match("(")==0)
293         error("missing opening parenthesis");
294     if (astreq(n+1,"main",4))
295         { hasmain=1; header(); }
296     outstr("_"); outstr(n); colon; nl;
297
298     locptr=startloc;
299     argstk=0;
300     while (match(")")==0)
301         {
302         if (symname(n))
303             {
304             if (findloc(n)) errmulti(n);
305             else
306                 {
307                 addloc(n,0,0,argstk);
308                 argstk=argstk+intwidth;
309                 }
310             }
311         else { error("illegal argument name");skipchars(); }
312         skipblanks();
313         if (streq(ch,")")==0)
314             {
315             if (match(",")==0) error("expected comma");
316             }
317         if (needstend()) break;
318         }
319     sp=0; argtop=argstk;
```

```
320     while (argstk)
321         {
322         if (amatch("char",4))
323             { getarg(cchar,argtop); needsemi(); }
324         else if (amatch("int",3))
325             { getarg(cint,argtop); needsemi(); }
326         else { error("wrong number of args"); break; }
327         }
328     if (statement()!=streturn)
329         {
330         modstk(0);
331         ret();
332         }
333     sp=0;
334     locptr=startloc;
335     }
336
337     /*********************************/
338     /*   Get function arguments      */
339     /*********************************/
340
341 getarg(t,argtop)
342     int t,argtop;
343     {
344     int j,legalname,address;
345     char n[namesize],c,*argptr;
346
347     while(1)
348         {
349         if (argstk==0) return;
350         if (match("*")) j=pointer; else j=variable;
351         if ((legalname=symname(n))==0) errname();
352         if (match("["))
353             {
354             while (inbyte()!=']')
355                 if (needstend()) break;
356             j=pointer;
357             }
358         if (legalname)
359             {
360             if (argptr=findloc(n))
361                 {
362                 argptr[ident]=j;
363                 argptr[type]=t;
364                 address=argtop-((argptr[offset]&255)+
365                     ((argptr[offset+1]&255)<<8));
366                 argptr[offset]=address;
367                 argptr[offset+1]=address>>8;
368                 }
369             else error("expecting argument name");
370         }
371         argstk=argstk-intwidth;
372         if (needstend()) return;
373         if (match(",")==0) error("expected comma");
374         }
```

```
375     }
376
377     /**********************************/
378     /*    Process a statement       */
379     /**********************************/
380
381 statement()
382     {
383     if ((*ch==0) & (eof)) return;
384     else if (amatch("char",4))
385         { declloc(cchar); needsemi(); }
386     else if (amatch("int",3))
387         { declloc(cint); needsemi(); }
388     else if (match("{"))
389         compound();
390     else if (amatch("if",2))
391         { doif(); laststst=stif; }
392     else if (amatch("while",5))
393         { dowhile(); laststst=stwhile; }
394     else if (amatch("return",6))
395         { doreturn(); needsemi(); laststst=streturn; }
396     else if (amatch("break",5))
397         { dobreak(); needsemi(); laststst=stbreak; }
398     else if (amatch("continue",8))
399         { docont(); needsemi(); laststst=stcont; }
400     else if (match(";"));
401     else if (match("$asm"))
402         { doasm(); laststst=stasm; }
403     else    {expression(); needsemi(); laststst=stexp; }
404     return laststst;
405     }
406
407     /**********************************/
408     /*  Process compound statement  */
409     /**********************************/
410
411 compound()
412     {
413     ++ncmp;
414     while (match("}")==0)
415         if (eof) return; else statement();
416     --ncmp;
417     }
418
419     /**********************************/
420     /*    Process IF statement      */
421     /**********************************/
422
423 doif()
424     {
425     int flev,fsp,flab1,flab2;
426
427     flev=locptr;
428     fsp=sp;
429     flab1=getlabel();
```

```
430     test(flab1);
431     statement();
432     sp=modstk(fsp);
433     locptr=flev;
434     if (amatch("else",4)==0)
435         {
436         outlabel(flab1); colon; nl;
437         return;
438         }
439     jump(flab2=getlabel());
440     outlabel(flab1); colon; nl;
441     statement();
442     sp=modstk(fsp);
443     locptr=flev;
444     outlabel(flab2); colon; nl;
445     }
446
447     /*********************************/
448     /*   Process WHILE statement     */
449     /*********************************/
450
451 dowhile()
452     {
453     int wq[4];
454
455     wq[wqsym]=locptr;
456     wq[wqsp]=sp;
457     wq[wqloop]=getlabel();
458     wq[wqlab]=getlabel();
459     addwhile(wq);
460     outlabel(wq[wqloop]); colon; nl;
461     test(wq[wqlab]);
462     statement();
463     jump(wq[wqloop]);
464     outlabel(wq[wqlab]); colon; nl;
465     locptr=wq[wqsym];
466     sp=modstk(wq[wqsp]);
467     delwhile();
468     }
469
470     /*********************************/
471     /*   Process RETURN statement    */
472     /*********************************/
473
474 doreturn()
475     {
476     if (needstend()==0) expression();
477     modstk(0);
478     ret();
479     }
480
481     /*********************************/
482     /*   Process BREAK statement     */
483     /*********************************/
484
```

```
485 dobreak()
486     {
487     int *ptr;
488
489     if ((ptr=readwhile())==0) return;
490     modstk((ptr[wqsp]));
491     jump(ptr[wqlab]);
492     }
493
494     /*********************************/
495     /*  Process CONTINUE statement  */
496     /*********************************/
497
498 docont()
499     {
500     int *ptr;
501
502     if ((ptr=readwhile())==0) return;
503     modstk((ptr[wqsp]));
504     jump(ptr[wqloop]);
505     }
506
507     /*********************************/
508     /*   Process #asm directive     */
509     /*********************************/
510
511 doasm()
512     {
513     cmode=0;
514     while(1)
515         {
516         inline();
517         if (match("#endasm")) break;
518         if (eof) break;
519         outstr(line);
520         nl;
521         }
522     resetptr();
523     cmode=1;
524     }
525
526     /*********************************/
527     /*   Expression evaluation by   */
528     /*      recursive descent       */
529     /*********************************/
530
531 expression()
532     {
533     int lval[2];
534
535     if (hier1(lval)) rvalue(lval);
536     }
537
538 hier1(lval)
539     int lval[];
```

```
540     {
541     int k, lval2[2];
542
543     k=hier2(lval);
544     if (match("="))
545         {
546         if (k==0) {errlval();return 0;}
547         if (lval[1]) push();
548         if (hier1(lval2)) rvalue(lval2);
549         store(lval);
550         return 0;
551         }
552     else return k;
553     }
554
555 hier2(lval)
556     int lval[];
557     {
558     int k, lval2[2];
559
560     k=hier3(lval);
561     skipblanks();
562     if (*ch!='|') return k;
563     if (k) rvalue(lval);
564     while(1)
565         {
566         if (match("|"))
567             {
568             push();
569             if (hier3(lval2)) rvalue(lval2);
570             pop();
571             or();
572             }
573         else return 0;
574         }
575     }
576
577 hier3(lval)
578     int lval[];
579     {
580     int k, lval2[2];
581
582     k=hier4(lval);
583     skipblanks();
584     if (*ch!='^') return k;
585     if (k) rvalue(lval);
586     while (1)
587         {
588         if (match("^"))
589             {
590             push();
591             if (hier4(lval2)) rvalue(lval2);
592             pop();
593             xor();
594             }
```

```
595         else return 0;
596         }
597     }
598
599 hier4(lval)
600     int lval[];
601     {
602     int k, lval2[2];
603
604     k=hier5(lval);
605     skipblanks();
606     if (*ch!='&') return k;
607     if (k) rvalue(lval);
608     while (1)
609         {
610         if (match("&"))
611             {
612             push();
613             if (hier5(lval2)) rvalue(lval2);
614             pop();
615             and();
616             }
617         else return 0;
618         }
619     }
620
621 hier5(lval)
622     int lval[];
623     {
624     int k, lval2[2];
625
626     k=hier6(lval);
627     skipblanks();
628     /* check for == and !=  */
629     if (*nexch != '=') return k;
630     if ((*ch != '=') & (*ch != '!')) return k;
631     if (k) rvalue(lval);
632     while (1)
633         {
634         if (match("=="))
635             {
636             push();
637             if (hier6(lval2)) rvalue(lval2);
638             pop();
639             eq();
640             }
641         else if (match("!="))
642             {
643             push();
644             if (hier6(lval2)) rvalue(lval2);
645             pop();
646             ne();
647             }
648         else return 0;
649         }
```

```
650     }
651
652 hier6(lval)
653     int lval[];
654     {
655     int k, lval2[2];
656
657     k=hier7(lval);
658     skipblanks();
659     /* wish to identify >, <, >=, <=, but reject >>, << */
660     if ( ( *ch    != '<') & ( *ch    != '>') ) return k;
661     if ( ( *nexch == '<') | ( *nexch == '>') ) return k;
662     if (k) rvalue(lval);
663     while (1)
664         {
665         if (match("<="))
666             {
667             push();
668             if (hier7(lval2)) rvalue(lval2);
669             pop();
670             if (cptr=lval[0])
671                 if (cptr[ident]==pointer)
672                     {
673                     ule();
674                     continue;
675                     }
676             if (cptr=lval2[0])
677                 if (cptr[ident]==pointer)
678                     {
679                     ule();
680                     continue;
681                     }
682             le();
683             }
684         else if (match(">="))
685             {
686             push();
687             if (hier7(lval2)) rvalue(lval2);
688             pop();
689             if (cptr=lval[0])
690                 if (cptr[ident]==pointer)
691                     {
692                     uge();
693                     continue;
694                     }
695             if (cptr=lval2[0])
696                 if (cptr[ident]==pointer)
697                     {
698                     uge();
699                     continue;
700                     }
701             ge();
702             }
703         else if ((streq(ch,"<")) & (streq(ch,"<<")==0))
704             {
```

```
705             inbyte();
706             push();
707             if (hier7(lval2)) rvalue(lval2);
708             pop();
709             if (cptr=lval[0])
710                 if (cptr[ident]==pointer)
711                     {
712                     ult();
713                     continue;
714                     }
715             if (cptr=lval2[0])
716                 if (cptr[ident]==pointer)
717                     {
718                     ult();
719                     continue;
720                     }
721             lt();
722             }
723         else if ((streq(ch,">")) & (streq(ch,">>")==0))
724             {
725             inbyte();
726             push();
727             if (hier7(lval2)) rvalue(lval2);
728             pop();
729             if (cptr=lval[0])
730                 if (cptr[ident]==pointer)
731                     {
732                     ugt();
733                     continue;
734                     }
735             if (cptr=lval2[0])
736                 if (cptr[ident]==pointer)
737                     {
738                     ugt();
739                     continue;
740                     }
741             gt();
742             }
743         else return 0;
744         }
745     }
746
747 hier7(lval)
748     int lval[];
749     {
750     int k, lval2[2];
751
752     k=hier8(lval);
753     skipblanks();
754     if ((streq(ch,">>")==0) & (streq(ch,"<<")==0)) return k;
755     if (k) rvalue(lval);
756     while (1)
757         {
758         if (match(">>"))
759             {
```

```
760             push();
761             if (hier8(lval2)) rvalue(lval2);
762             pop();
763             asr();
764             }
765         else if (match("<<"))
766             {
767             push();
768             if (hier8(lval2)) rvalue(lval2);
769             pop();
770             asl();
771             }
772         else return 0;
773         }
774     }
775
776 hier8(lval)
777     int lval[];
778     {
779     int k, lval2[2];
780
781     k=hier9(lval);
782     skipblanks();
783     if ((*ch!='+') & (*ch!='-')) return k;
784     if (k) rvalue(lval);
785     while (1)
786         {
787         if (match("+"))
788             {
789             push();
790             if (hier9(lval2)) rvalue(lval2);
791             if (cptr=lval[0])
792                 if ((cptr[ident]==pointer) &
793                     (cptr[type]==cint)) scale(intwidth);
794             pop();
795             add();
796             }
797         else if (match("-"))
798             {
799             push();
800             if (hier9(lval2)) rvalue(lval2);
801             if (cptr=lval[0])
802                 if ((cptr[ident]==pointer) &
803                     (cptr[type]==cint)) scale(intwidth);
804             pop();
805             sub();
806             }
807         else return 0;
808         }
809     }
810
811 hier9(lval)
812     int lval[];
813     {
814     int k, lval2[2];
```

```
815
816     k=hier10(lval);
817     skipblanks();
818     if ((*ch!='*') & (*ch!='/') & (*ch!='%')) return k;
819     if (k) rvalue(lval);
820     while (1)
821         {
822         if (match("*"))
823             {
824             push();
825             if (hier9(lval2)) rvalue(lval2);
826             pop();
827             mult();
828             }
829         else if (match("/"))
830             {
831             push();
832             if (hier10(lval2)) rvalue(lval2);
833             pop();
834             div();
835             }
836         else if (match("%"))
837             {
838             push();
839             if (hier10(lval2)) rvalue(lval2);
840             pop();
841             mod();
842             }
843         else return 0;
844         }
845     }
846
847 hier10(lval)
848     int lval[];
849     {
850     int k;
851     char *ptr;
852
853     if (match("++"))
854         {
855         if ((k=hier10(lval))==0) { errlval(); return 0;}
856         if (lval[1]) push();
857         rvalue(lval);
858         ptr=lval[0];
859         if ((ptr[ident]==pointer) & (ptr[type]==cint))
860              inc(intwidth);
861         else inc(charwidth);
862         store(lval);
863         return 0;
864         }
865     else if (match("--"))
866         {
867         if ((k=hier10(lval))==0) { errlval(); return 0;}
868         if (lval[1]) push();
869         rvalue(lval);
```

```
870         ptr=lval[0];
871         if ((ptr[ident]==pointer) & (ptr[type]==cint))
872              dec(intwidth);
873         else dec(charwidth);
874         store(lval);
875         return 0;
876         }
877     else if (match("-"))
878         {
879         k=hier10(lval);
880         if (k) rvalue(lval);
881         neg();
882         return 0;
883         }
884     else if (match("*"))
885         {
886         k=hier10(lval);
887         if (k) rvalue(lval);
888         lval[1]=cint;
889         if (ptr=lval[0]) lval[1]=ptr[type];
890         lval[0]=0;
891         return 1;
892         }
893     else if (match("&"))
894         {
895         k=hier10(lval);
896         if (k==0) { error("illegal address"); return 0; }
897         else if (lval[1]) return 0;
898              else
899             {
900             immed();
901             outstr(ptr=lval[0]);
902             nl;
903             lval[1]=ptr[type];
904             return 0;
905             }
906         }
907     else
908         {
909         k=hier11(lval);
910         if (match("++"))
911             {
912             if (k==0) { errlval();return 0; }
913             if (lval[1]) push();
914             rvalue(lval);
915             ptr=lval[0];
916             if ((ptr[ident]==pointer) & (ptr[type]==cint))
917                  inc(intwidth);
918             else inc(charwidth);
919             store(lval);
920             if ((ptr[ident]==pointer) & (ptr[type]==cint))
921                  dec(intwidth);
922             else dec(charwidth);
923             return 0;
924             }
```

```
925         else if (match("--"))
926             {
927             if (k==0) { errlval(); return 0; }
928             if (lval[1]) push();
929             rvalue(lval);
930             ptr=lval[0];
931             if ((ptr[ident]==pointer) & (ptr[type]==cint))
932                  dec(intwidth);
933             else dec(charwidth);
934             store(lval);
935             if ((ptr[ident]==pointer) & (ptr[type]==cint))
936                  inc(intwidth);
937             else inc(charwidth);
938             return 0;
939             }
940         else return k;
941         }
942     }
943
944 hier11(lval)
945     int *lval;
946     {
947     int k;
948     char *ptr;
949
950     k=primary(lval);
951     ptr=lval[0];
952     skipblanks();
953     if ((*ch=='[') | (*ch=='('))
954         while (1)
955             {
956             if (match("["))
957                 {
958                 if (ptr==0)
959                     {
960                     error("can't subscript");
961                     skipchars();
962                     needbrack("]");
963                     return 0;
964                     }
965                 else if (ptr[ident]==pointer)
966                     rvalue(lval);
967                 else if (ptr[ident]!=array)
968                     {
969                     error("can't subscript");
970                     k=0;
971                     }
972                 push();
973                 expression();
974                 needbrack("]");
975                 if (ptr[type]==cint) scale(intwidth);
976                 pop();
977                 add();
978                 lval[0]=0;
979                 lval[1]=ptr[type];
```

```
 980                 k=1;
 981                 }
 982             else if (match("("))
 983                 {
 984                 if (ptr==0) callfunction(0);
 985                 else if (ptr[ident]!=function)
 986                     {
 987                     rvalue(lval);
 988                     callfunction(0);
 989                     }
 990                 else callfunction(ptr);
 991                 k=lval[0]=0;
 992                 }
 993             else return k;
 994             }
 995         if (ptr==0) return k;
 996         if (ptr[ident]==function)
 997             {
 998             immed();
 999             outstr(ptr);
1000             nl;
1001             return 0;
1002             }
1003         return k;
1004     }
1005
1006 primary(lval)
1007     int *lval;
1008     {
1009     char *ptr, sname[namesize];
1010     int num[1];
1011     int k;
1012
1013     if (match("("))
1014         {
1015         k=hier1(lval);
1016         needbrack(")");
1017         return k;
1018         }
1019     if (symname(sname))
1020         {
1021         if (ptr=findloc(sname))
1022             {
1023             getloc(ptr);
1024             lval[0]=ptr;
1025             lval[1]=ptr[type];
1026             if (ptr[ident]==pointer) lval[1]=cint;
1027             if (ptr[ident]==array) return 0; else return 1;
1028             }
1029         if (ptr=findglb(sname)) if (ptr[ident]!=function)
1030                     {
1031                     lval[0]=ptr;
1032                     lval[1]=0;
1033                     if (ptr[ident]!=array) return 1;
1034                     immed();
```

```
1035                    outstr(ptr);
1036                    nl;
1037                    lval[1]=ptr[type];
1038                    return 0;
1039                    }
1040        ptr=addglb(sname,function,cint,0);
1041        lval[0]=ptr;
1042        lval[1]=0;
1043        return 0;
1044        }
1045    if (constant(num)) return(lval[0]=lval[1]=0);
1046    else
1047        {
1048        error("invalid expression");
1049        immed();
1050        outdec(0);
1051        nl;
1052        skipchars();
1053        return 0;
1054        }
1055    }
1056
1057    /*********************************/
1058    /*   Process function call       */
1059    /*********************************/
1060
1061 callfunction(ptr)
1062    char *ptr;
1063    {
1064    int nargs;
1065
1066    nargs=0;
1067    skipblanks();
1068    if (ptr==0) push();
1069    while (*ch != ')')
1070        {
1071        if (needstend()) break;
1072        expression();
1073        if (ptr==0) swapstk();
1074        push();
1075        nargs=nargs+intwidth;
1076        if (match(",")==0) break;
1077        }
1078    needbrack(")");
1079    if (ptr) call(ptr+1);
1080    else callstk();
1081    sp=modstk(sp+nargs);
1082    }
1083
1084    /*********************************/
1085    /* Declare a static variable -   */
1086    /* makes an entry in the symbol */
1087    /* table so that subsequent      */
1088    /* references can call symbol    */
1089    /* by name                       */
```

```
1090     /*******************************/
1091
1092 declglb(typ)
1093     int typ;
1094     {
1095     int k,j; char sname[namesize];
1096
1097     while(1)
1098         {
1099         while(1)
1100             {
1101             if (needstend()) return;
1102             k=1;
1103             if (match("*")) j=pointer;
1104             else j=variable;
1105             if (symname(sname)==0) errname();
1106             if (findglb(sname)) errmulti(sname);
1107             if (match("["))
1108                 {
1109                 k=needsub();
1110                 if (k) j=array;
1111                 else j=pointer;
1112                 }
1113             addglb(sname,j,typ,k);
1114             break;
1115             }
1116         if (match(",")==0) return;
1117         }
1118     }
1119
1120     /*******************************/
1121     /*   Declare local variables   */
1122     /*******************************/
1123
1124 declloc(typ)
1125     int typ;
1126     {
1127     int k,j; char sname[namesize];
1128
1129     while(1)
1130         {
1131         while(1)
1132             {
1133             if (needstend()) return;
1134             if (match("*")) j=pointer;
1135             else j=variable;
1136             if (symname(sname)==0) errname();
1137             if (findloc(sname)) errmulti(sname);
1138             if (match("["))
1139                 {
1140                 k= needsub();
1141                 if (k)
1142                     {
1143                     j=array;
1144                     if (typ==cint) k=k*intwidth;
```

```
1145                    }
1146                else
1147                    {
1148                    j=pointer;
1149                    k=intwidth;
1150                    }
1151                }
1152            else
1153                if ((typ==cchar) & (j!=pointer)) k=charwidth;
1154                else k=intwidth;
1155            sp=modstk(sp-k);
1156            addloc(sname,j,typ,sp);
1157            break;
1158            }
1159        if (match(",")==0) return;
1160        }
1161    }
1162
1163    /*********************************/
1164    /*   Insert new global symbol    */
1165    /*********************************/
1166
1167 addglb(sname,id,typ,value)
1168     char *sname,id,typ;
1169     int value;
1170     {
1171     char *ptr;
1172
1173     if (cptr=findglb(sname)) return cptr;
1174     if (glbptr>=endglb)
1175         {
1176         error("global symbol table overflow");
1177         return 0;
1178         }
1179     cptr=ptr=glbptr;
1180     while (*ptr++ = *sname++);
1181     cptr[ident]=id;
1182     cptr[type]=typ;
1183     cptr[storage]=statik;
1184     cptr[offset]=value;
1185     cptr[offset+1]=value>>8;
1186     glbptr=glbptr+symsiz;
1187     return cptr;
1188     }
1189
1190     /*********************************/
1191     /*  Find a global symbol name    */
1192     /*********************************/
1193
1194 findglb(sname)
1195     char *sname;
1196     {
1197     char *ptr;
1198
1199     ptr=startglb;
```

```
1200     while (ptr!=glbptr)
1201         {
1202         if (*sname == *ptr)        /* check lengths */
1203             if (streq(sname,ptr)) return ptr;
1204         ptr=ptr+symsiz;
1205         }
1206     return 0;
1207     }
1208
1209     /*********************************/
1210     /*    Insert new local symbol    */
1211     /*********************************/
1212
1213 addloc(sname,id,typ,value)
1214     char *sname,id,typ;
1215     int value;
1216     {
1217     char *ptr;
1218
1219     if (cptr=findloc(sname)) return cptr;
1220     if (locptr>=endloc)
1221         {
1222         error("local symbol table overflow");
1223         return 0;
1224         }
1225     cptr=ptr=locptr;
1226     while (*ptr++ = *sname++);
1227     cptr[ident]=id;
1228     cptr[type]=typ;
1229     cptr[storage]=stkloc;
1230     cptr[offset]=value;
1231     cptr[offset+1]=value>>8;
1232     locptr=locptr+symsiz;
1233     return cptr;
1234     }
1235
1236     /*********************************/
1237     /*  Find a local symbol name     */
1238     /*********************************/
1239
1240 findloc(sname)
1241     char *sname;
1242     {
1243     char *ptr;
1244
1245     ptr=startloc;
1246     while (ptr!=locptr)
1247         {
1248         if (*sname == *ptr)        /* check lengths */
1249             if (streq(sname,ptr)) return ptr;
1250         ptr=ptr+symsiz;
1251         }
1252     return 0;
1253     }
1254
```

```
1255     /*********************************/
1256     /*  Put a new macro definition  */
1257     /*         in the table          */
1258     /*********************************/
1259
1260 addmac()
1261     {
1262     char sname[namesize], *sn, *mn;
1263
1264     if (symname(sname)==0)
1265         {
1266         errname();
1267         resetptr();
1268         return;
1269         }
1270     sn=sname; mn=mactptr;
1271     if  (macpmax >= macpptr)
1272         {               /* add macro to table */
1273         while (*mn++ = *sn++);
1274         *macpptr++ = macbptr; mactptr = mactptr + namesize;
1275         }
1276     else error("macro count exceeded");
1277     while (*ch==' ' | *ch==tabch) gch();
1278     while (*macbptr++ = gch()) /* add macro body to buffer */
1279         {
1280         if (macbptr>=macbmax)
1281             {error("macro table full");break;}
1282         }
1283     }
1284
1285     /*********************************/
1286     /* Look up possible macro name  */
1287     /*********************************/
1288
1289 findmac(sname)
1290     char *sname;
1291     {
1292     int *mqp;
1293     char *mqt;
1294
1295     mqp=macp; mqt=mact;
1296     while (mqp<macpptr)
1297         {
1298         if (*sname == *mqt)     /* check lengths */
1299             if (streq(sname,mqt)) return (*mqp);
1300         mqt=mqt+namesize; mqp++;
1301         }
1302     return 0;
1303     }
1304
1305     /*********************************/
1306     /*  WHILE table manipulations   */
1307     /*********************************/
1308
1309 addwhile(ptr)
```

```
1310     int ptr[];
1311     {
1312     int k;
1313
1314     if (wqptr==wqmax)
1315         { error("too many active whiles"); return; }
1316     k=0;
1317     while (k<wqsiz)
1318         { *wqptr++=ptr[k++]; }
1319     }
1320
1321 delwhile()
1322     {
1323     if (readwhile()) wqptr=wqptr-wqsiz;
1324     }
1325
1326 readwhile()
1327     {
1328     if (wqptr==wq)
1329         { error("no active whiles"); return 0; }
1330     else return (wqptr-wqsiz);
1331     }
1332
1333     /*********************************/
1334     /*     Generate next label       */
1335     /*********************************/
1336
1337 getlabel()
1338     {
1339     return(++nxtlab);
1340     }
1341
1342     /*********************************/
1343     /*       Read symbol name        */
1344     /*********************************/
1345
1346 symname(sname)
1347     char *sname;
1348     {
1349     int k;
1350
1351     skipblanks();
1352     if (alpha(*ch)==0) return 0;
1353     k=1;
1354     while (an(*ch)) sname[k++]=gch();
1355     sname[k]=0;
1356     sname[0]=k-1; /* first 'char' is length of symbol */
1357     return 1;
1358     }
1359
1360     /*********************************/
1361     /* Check for a number in input  */
1362     /*********************************/
1363
1364 number(val)
```

```
1365      int val[];
1366      {
1367      int k, minus;
1368      char c;
1369
1370      k=minus=1;
1371      while (k)
1372          {
1373          k=0;
1374          if (match("+")) k=1;
1375          if (match("-")) { minus=(-minus); k=1; }
1376          }
1377      if (numeric(*ch)==0) return 0;
1378      while (numeric(*ch))
1379          {
1380          c=inbyte();
1381          k=k*10+(c-'0');
1382          }
1383      if (minus<0) k=(-k);
1384      val[0]=k;
1385      return 1;
1386      }
1387
1388      /*********************************/
1389      /*       Load a constant         */
1390      /*********************************/
1391
1392  constant(val)
1393      int val[];
1394      {
1395      if (number(val)) immed();
1396      else if (getqchar(val)) immed();
1397          else if (getqstring(val))
1398                  {
1399                  immed();
1400                  outlabel(litlab);
1401                  outbyte('+');
1402                  }
1403              else return 0;
1404      outdec(val[0]);
1405      nl;
1406      return 1;
1407      }
1408
1409      /*********************************/
1410      /*  Get one or two characters    */
1411      /*     from input stream         */
1412      /*********************************/
1413
1414  getqchar(val)
1415      int val[];
1416      {
1417      int k;
1418      char c;
1419
```

```
1420     k=0;
1421     if (match("'")==0) return 0;
1422     if ( (c=gch())==bslch )     /* escape sequence? */
1423         {
1424         if ( (c=gch())=='n') k=eolch;  /* newline   */
1425         else if (c=='t') k=tabch;      /* tab       */
1426         else if (c=='b') k=bspch;      /* backspace */
1427         else if (c=='r') k=crch;       /* return    */
1428         else if (c=='f') k=ffch;       /* form feed */
1429         else if (c=='\\') k=bslch;     /* backslash */
1430         else if (c=='0') k=0;          /* null      */
1431         else k=c;
1432         }
1433     else k=c;
1434     if (match("'")==0) return 0;
1435     val[0]=k;
1436     return 1;
1437     }
1438
1439     /*********************************/
1440     /* Get string from input stream */
1441     /*********************************/
1442
1443 getqstring(val)
1444     int val[];
1445     {
1446     char c;
1447
1448     if (match(quote)==0) return 0;
1449     val[0]=litptr;
1450     while ( *ch!='"' )
1451         {
1452         if ( *ch==0 ) break;
1453         if (litptr >= litmax)
1454             {
1455             error("string space exhausted");
1456             while (match(quote)==0) if (gch()==0) break;
1457             return 1;
1458             }
1459         litq[litptr++]=gch();
1460         }
1461     gch();
1462     litq[litptr++]=0;
1463     return 1;
1464     }
1465
1466     /*********************************/
1467     /*       String compare          */
1468     /*********************************/
1469
1470 streq(str1, str2)
1471     char *str1, *str2;
1472     {
1473     int k;
1474
```

```
1475      k=0;
1476      while (*str2)
1477          {
1478          if ((*str1)!=(*str2)) return 0;
1479          k++; str1++; str2++;
1480          }
1481      return k;
1482      }
1483
1484      /*********************************/
1485      /*  String compare over `len'    */
1486      /*          characters           */
1487      /*********************************/
1488
1489  astreq(str1, str2, len)
1490      char *str1, *str2;
1491      int  len;
1492      {
1493      int k;
1494
1495      k=0;
1496      while (k<len)
1497          {
1498          if ((*str1)!=(*str2)) break;
1499          if (*str1==0) break;
1500          if (*str2==0) break;
1501          k++; str1++; str2++;
1502          }
1503      if (an(*str1)) return 0;
1504      if (an(*str2)) return 0;
1505      return k;
1506      }
1507
1508      /*********************************/
1509      /*  Compare literal with line    */
1510      /*  buffer contents, advancing   */
1511      /*   buffer pointer if found     */
1512      /*********************************/
1513
1514  match(lit)
1515      char *lit;
1516      {
1517      int k;
1518
1519      skipblanks();
1520      if (k=streq(ch,lit))
1521          { ch=ch+k; nexch=ch+1; return 1; }
1522      return 0;
1523      }
1524
1525      /*********************************/
1526      /*  As `match', but over `len'   */
1527      /*          characters           */
1528      /*********************************/
1529
```

```
1530 amatch(lit, len)
1531     char *lit;
1532     int len;
1533     {
1534     int k;
1535
1536     skipblanks();
1537     if (k=astreq(ch,lit,len))
1538         {
1539         ch=ch+k; nexch=ch+1;
1540         while (an(*ch)) inbyte();
1541         return 1;
1542         }
1543     return 0;
1544     }
1545
1546     /*********************************/
1547     /*     Get array bounds          */
1548     /*********************************/
1549
1550 needsub()
1551     {
1552     int   num[1];
1553
1554     if (match("]")) return 0;
1555     if (number(num)==0)
1556         {
1557         error("must be constant");
1558         num[0]=1;
1559         }
1560     if (num[0]<0)
1561         {
1562         error("negative size illegal");
1563         num[0]=(-num[0]);
1564         }
1565     needbrack("]");
1566     return num[0];
1567     }
1568
1569     /*********************************/
1570     /*    Check for semicolon        */
1571     /*********************************/
1572
1573 needsemi()
1574     {
1575     if (match(";")==0) error("missing semicolon");
1576     }
1577
1578     /*********************************/
1579     /*  Check for end of statement   */
1580     /*********************************/
1581
1582 needstend()
1583     {
1584     skipblanks();
```

```
1585     return ((streq(ch,";") | (*ch==0)));
1586     }
1587
1588 needbrack(str)
1589     char *str;
1590     {
1591     if (match(str)==0)
1592         {
1593         error("missing bracket");
1594         comment; outstr(str); nl;
1595         }
1596     }
1597
1598     /*********************************/
1599     /*  Skip white space in input    */
1600     /*********************************/
1601
1602 skipblanks()
1603     {
1604     while (1)
1605         {
1606         while (*ch==0)
1607             {
1608             inline();
1609             preprocess();
1610             if (eof) break;
1611             }
1612         if (*ch==' ') gch();
1613         else if (*ch==tabch) gch(); else return;
1614         }
1615     }
1616
1617     /*********************************/
1618     /*   Skip this token and all     */
1619     /*   succeeding white space      */
1620     /*********************************/
1621
1622 skipchars()
1623     {
1624     if (an(inbyte())) while (an(*ch)) gch();
1625     else while (an(*ch)==0)
1626         { if (*ch==0) break; gch(); }
1627     skipblanks();
1628     }
1629
1630     /*********************************/
1631     /*      Test input character     */
1632     /*********************************/
1633
1634 alpha(c)
1635     char c;
1636     {
1637     if ( (c >= 'a')&(c <= 'z') ) return 1;
1638     if ( (c >= 'A')&(c <= 'Z') ) return 1;
1639     return  (c=='_') ;
```

```
1640     }
1641
1642 numeric(c)
1643     char c;
1644     {
1645     return((c>='0')&(c<='9'));
1646     }
1647
1648 an(c)
1649     char c;
1650     {
1651     if ( (c >= '0')&(c <= '9') ) return 1;
1652     if ( (c >= 'a')&(c <= 'z') ) return 1;
1653     if ( (c >= 'A')&(c <= 'Z') ) return 1;
1654     return  (c=='_') ;
1655     }
1656
1657     /*********************************/
1658     /* Get a character - read in a  */
1659     /* line and preprocess if you   */
1660     /*          have to             */
1661     /*********************************/
1662
1663 inbyte()
1664     {
1665     while (*ch==0)
1666         {
1667         if (eof) return 0;
1668         inline();
1669         preprocess();
1670         }
1671     return gch();
1672     }
1673
1674     /*********************************/
1675     /* Get a character - read in a  */
1676     /*   line if you have to, but   */
1677     /* don't preprocess (since this */
1678     /* is called from preprocessor!)*/
1679     /*********************************/
1680
1681 inchar()
1682     {
1683     if (*ch==0) inline();
1684     if (eof) return 0;
1685     return (gch());
1686     }
1687
1688     /*********************************/
1689     /*       Read in a line          */
1690     /*********************************/
1691
1692 inline()
1693     {
1694     int k,unit;
```

```
1695
1696     while(1)
1697         {
1698         if (input==0)
1699             if ( mulfile) openin(); else eof=1;
1700         if (eof) return;
1701         if ((unit=input2)==0) unit=input;
1702         resetptr();
1703         while ((k=getc(unit))>0)
1704             {
1705             if ((k==eolch) | (ch>=chmax))
1706                 { lncnt++; break; }
1707             *ch++=k;
1708             }
1709         *ch=0;
1710         if (k<=0)
1711             {
1712             fclose(unit);
1713             if (input2) input2=0; else input=0;
1714             }
1715         if (ch>line)
1716             {
1717             if ((ctext)&(cmode))
1718                 {
1719                 comment;
1720                 outstr(line);
1721                 nl;
1722                 }
1723             ch=line; nexch=ch+1;
1724             return;
1725             }
1726         }
1727     }
1728
1729     /*********************************/
1730     /*  Preprocess a line - expand  */
1731     /* macros and remove redundant  */
1732     /*      tabs and spaces         */
1733     /*********************************/
1734
1735 preprocess()
1736     {
1737     int k;
1738     char c,sname[namesize],*mq;
1739
1740     if (cmode==0) return;
1741     mptr=mline; ch=line; nexch=ch+1;
1742     while (*ch)
1743         {
1744         if ((*ch==' ') | (*ch==tabch))
1745             {
1746             keepch(' ');
1747             while ((*ch==' ') | (*ch==tabch)) gch();
1748             }
1749         else if (*ch=='"')
```

```
1750            {
1751            keepch(*ch);
1752            gch();
1753            while (*ch!='"')
1754                {
1755                if (*ch==0)
1756                    {error("missing quote");break;}
1757                keepch(gch());
1758                }
1759            gch();
1760            keepch('"');
1761            }
1762        else if (*ch==quoch)
1763            {
1764            keepch(quoch);
1765            gch();
1766            while (*ch!=quoch)
1767                {
1768                if (*ch==0)
1769                    {error("missing apostrophe");break;}
1770                keepch(gch());
1771                }
1772            gch();
1773            keepch(quoch);
1774            }
1775        else if ((*ch=='/') & (*nexch=='*'))
1776            {
1777            inchar(); inchar();
1778            while (((*ch=='*') & (*nexch=='/'))==0)
1779                {
1780                if (*ch==0) inline(); else inchar();
1781                if (eof) break;
1782                }
1783            inchar(); inchar();
1784            }
1785        else if (an(*ch))
1786            {
1787            k=1;
1788            while (an(*ch))
1789                {
1790                if (k<namemax) sname[k++]= *ch;
1791                gch();
1792                }
1793            sname[k]=0; sname[0]=k-1;
1794            if (mq=findmac(sname))
1795                while (c= *mq++) keepch(c);
1796            else
1797                {
1798                k=1;
1799                while (c=sname[k++]) keepch(c);
1800                }
1801            }
1802        else keepch(gch());
1803        }
1804    keepch(0);
```

```
1805     if (mptr>=mpmax) error("line too long");
1806     mptr=mline; ch=line; nexch=ch+1;
1807
1808     /* copy back to line buffer and strip parity for keeps */
1809
1810     while (*ch++= *mptr++&127);
1811     ch=line;
1812     }
1813
1814     /*********************************/
1815     /*      Reset input buffer       */
1816     /*********************************/
1817
1818 resetptr()
1819     {
1820     ch=line; nexch=ch+1; *ch=0;
1821     }
1822
1823     /*********************************/
1824     /*    Get next input character   */
1825     /*********************************/
1826
1827 gch()
1828     {
1829     if (*ch==0) return 0;
1830     else { nexch++; return (*ch++);}
1831     }
1832
1833     /*********************************/
1834     /* Save this character in buffer*/
1835     /*********************************/
1836
1837 keepch(c)
1838     char c;
1839     {
1840     *mptr=c;
1841     if (mptr<mpmax) mptr++;
1842     return c;
1843     }
1844
1845     /*********************************/
1846     /*    Get options from user      */
1847     /*********************************/
1848
1849 ask()
1850     {
1851     resetptr();          /* clear input line     */
1852     clearscreen;         /* clear the screen     */
1853     display("         RatC : Lancaster implementation");
1854     display("         ------------------------------");
1855     nl; nl;
1856     glbflag=1;  nxtlab=1;  mulfile=0;
1857     display("Defaults:");
1858     display("        Globals defined   y");
1859     display("            First label   1");
```

```
1860     display("        Multiple files   n");
1861     nl;
1862     display("       Change defaults ? ");
1863     if (reply()) defaults();
1864
1865     litlab=getlabel();  /* first label = literal pool */
1866
1867             /* see if user wants to interleave c-text */
1868     ctext=0;
1869     display("      C-text to appear ? ");
1870     ctext=reply();
1871
1872     resetptr();                          /* erase line */
1873     }
1874
1875 defaults()
1876     {
1877     int k,num[1];
1878
1879           /* see if user wants us to allocate static  */
1880           /* variables by name in this module         */
1881           /* (pseudo external capability)             */
1882     display("     Globals to be defined ? ");
1883     glbflag=reply();
1884
1885       /* get first allowable # for compiler-generated */
1886       /* labels (so user can append modules)          */
1887     while(1)
1888         {
1889         display("Starting number for labels ? ");
1890         gets(line);
1891         if (*ch==0) {num[0]=0; break;}
1892         if (k=number(num)) break;
1893         }
1894     nxtlab=num[0];
1895
1896                    /* find out if one input file only */
1897     display("      Multiple input files ? ");
1898     mulfile=reply();
1899     }
1900
1901     /*********************************/
1902     /* Print a string to console     */
1903     /*********************************/
1904
1905 display(str)
1906     char *str;
1907     {
1908     int k;
1909
1910     k=0;
1911     putchar(eolch);
1912     while (str[k]) putchar(str[k++]);
1913     }
1914
```

```
1915 reply()
1916     {
1917     resetptr();      /* clear input buffer */
1918     gets(line);
1919     if ((*ch=='Y') | (*ch=='y')) return 1; else return 0;
1920     }
1921
1922     /*********************************/
1923     /*      Get output filename      */
1924     /*********************************/
1925
1926 openout()
1927     {
1928     resetptr();      /* erase line       */
1929     output=0;        /* start with none  */
1930     display("        Output filename ? ");
1931     gets(line);
1932     if (*ch==0) return;   /* none given */
1933     if ((output=fopen(line,"w"))==NULL)
1934         { output=0; error("Open failure"); }
1935     resetptr();
1936     }
1937
1938     /*********************************/
1939     /*      Get (next) input file    */
1940     /*********************************/
1941
1942 openin()
1943     {
1944     input=0;         /* none to start with   */
1945     while (input==0)
1946         {
1947         resetptr();
1948         if (eof) break;
1949         display("              Input filename ? ");
1950         gets(line);
1951         if (*ch==0) { eof=1; break; }
1952         if ((input=fopen(line,"r"))==NULL)
1953             { input=0; display("Open failure"); }
1954         }
1955     resetptr();
1956     }
1957
1958     /*********************************/
1959     /*      Open an include file     */
1960     /*********************************/
1961
1962 openinclude()
1963     {
1964     skipblanks();
1965     if ((input2=fopen(ch,"r"))==NULL)
1966         { input2=0; error("Open failure on include file"); }
1967     resetptr();
1968     }
1969
```

```
1970     /*********************************/
1971     /*     Close the output file     */
1972     /*********************************/
1973
1974 closeout()
1975     {
1976     if (output) fclose(output);
1977     output=0;
1978     }
1979
1980     /*********************************/
1981     /*       Error reporting         */
1982     /*********************************/
1983
1984 errlval()
1985     {
1986     error("must be lvalue");
1987     }
1988
1989 errmulti(sname)
1990     char *sname;
1991     {
1992     error("already defined");
1993     comment;
1994     outstr(sname); nl;
1995     }
1996
1997 errname()
1998     {
1999     error("illegal symbol name"); skipchars();
2000     }
2001
2002     /*********************************/
2003     /* Report type and position of   */
2004     /* an error in the source line   */
2005     /*********************************/
2006
2007 error(ptr)
2008     char ptr[];
2009     {
2010     char *k;
2011
2012     comment;outstr(line);nl;comment;
2013     k=line;
2014     while (k<ch)
2015         {
2016         if (*k==tabch) tab; else space;
2017         ++k;
2018         }
2019     outbyte('^');
2020     nl; comment; outstr("------ ");
2021     outstr(ptr);
2022     outstr(" ------");
2023     display(line); display(ptr);  /* to console too */
2024     nl;
```

```
2025      ++errcnt;
2026      }
2027
2028      /*********************************/
2029      /*    Report errors for user     */
2030      /*********************************/
2031
2032  errorsummary()
2033      {
2034      if (ncmp) error("missing closing bracket");
2035          /* open compound statement  */
2036      nl;
2037      outdec(errcnt);      /* total # errors   */
2038      outstr(" errors in compilation.");
2039      nl;
2040
2041      nl;
2042      outdec(inscnt); /* total # instructions */
2043      outstr(" instructions generated.");
2044      nl;
2045
2046      nl;
2047      outdec(lncnt);  /* total # source lines */
2048      outstr(" source lines.");
2049      nl;
2050      }
2051
2052      /*********************************/
2053      /*      Output a character       */
2054      /*********************************/
2055
2056  outbyte(c)
2057      char c;
2058      {
2059      if (c==0) return 0;
2060      if (output)
2061          {
2062          if ((putc(c,output))<=0)
2063              { closeout(); error("output file error"); }
2064          }
2065      else putchar(c);
2066      return c;
2067      }
2068
2069      /*********************************/
2070      /* Output a number in decimal    */
2071      /*********************************/
2072
2073  outdec(num)
2074      int num;
2075      {
2076      int k, zs;
2077      char c;
2078
2079      zs=0;
```

```
2080     k=10000;
2081     if (num<0)
2082         {
2083         num=(-num);
2084         outbyte('-');
2085         }
2086     while (k>=1)
2087         {
2088         c=num/k+'0';
2089         if ((c!='0')||(k==1)||(zs)) { zs=1; outbyte(c); }
2090         num=num%k;
2091         k=k/10;
2092         }
2093     }
2094
2095     /*********************************/
2096     /*      Output a string          */
2097     /*********************************/
2098
2099 outstr(ptr)
2100     char ptr[];
2101     {
2102     int k;
2103
2104     /* ignore length byte if there is one */
2105     if (ptr[0]<32) k=1; else k=0;
2106     while (outbyte(ptr[k++]));
2107     }
2108
2109     /*********************************/
2110     /*       Print a label           */
2111     /*********************************/
2112
2113 outlabel(label)
2114     int label;
2115     {
2116     outstr("cc");
2117     outdec(label);
2118     }
2119
2120     /*********************************/
2121     /*     Instruction output        */
2122     /*********************************/
2123
2124 outline(ptr)
2125     char ptr[];
2126     { outtab(ptr); nl; }
2127
2128 outtab(ptr)
2129     char ptr[];
2130     { tab; outstr(ptr); inscnt++; }
2131
2132     /*********************************/
2133     /*     Evaluate condition        */
2134     /*********************************/
```

```
2135
2136 test(label)
2137     int label;
2138     {
2139     needbrack("(");
2140     expression();
2141     needbrack(")");
2142     testjump(label);
2143     nl;
2144     }
2145
2146     /**********************************/
2147     /*    Store a value in memory   */
2148     /**********************************/
2149
2150 store(lval)
2151     int *lval;
2152     {
2153     if (lval[1]==0) putmem(lval[0]);
2154     else putstk(lval[1]);
2155     }
2156
2157     /**********************************/
2158     /*    Get a value from memory   */
2159     /**********************************/
2160
2161 rvalue(lval)
2162     int *lval;
2163     {
2164     if ((lval[0]!=0) & (lval[1]==0)) getmem(lval[0]);
2165     else indirect(lval[1]);
2166     }
2167
2168     /**********************************/
2169     /*   Load direct 8 or 16 bits   */
2170     /*    into primary register     */
2171     /**********************************/
2172
2173 getmem(sym)
2174     char *sym;
2175     {
2176     if ((sym[ident]!=pointer) & (sym[type]==cchar))
2177         outtab("ldir.b");
2178     else outtab("ldir.w");
2179     outline(sym+name+1);
2180     }
2181
2182     /**********************************/
2183     /* Given offset from SP, get    */
2184     /* address into primary register*/
2185     /**********************************/
2186
2187 getloc(sym)
2188     char *sym;
2189     {
```

```
2190      outtab("addr"); tab;
2191      outdec(((sym[offset]&255)+((sym[offset+1]&255)<<8))-sp);
2192      nl;
2193      }
2194
2195      /*********************************/
2196      /*  Store direct 8 or 16 bits    */
2197      /*    from primary register      */
2198      /*********************************/
2199
2200  putmem(sym)
2201      char *sym;
2202      {
2203      if ((sym[ident]!=pointer) & (sym[type]==cchar))
2204          outtab("sdir.b");
2205      else outtab("sdir.w");
2206      tab; outstr(sym+name);
2207      nl;
2208      }
2209
2210      /*********************************/
2211      /* Store indirect 8 or 16 bits   */
2212      /* at address on top of stack    */
2213      /*********************************/
2214
2215  putstk(typeobj)
2216      char typeobj;
2217      {
2218      pop();
2219      if (typeobj==cchar) outline("sind.b");
2220      else outline("sind.w");
2221      }
2222
2223      /*********************************/
2224      /*  Load indirect 8 or 16 bits   */
2225      /*  at address in primary reg    */
2226      /*    into primary register      */
2227      /*********************************/
2228
2229  indirect(typeobj)
2230      char typeobj;
2231      {
2232      if (typeobj==cchar) outline("lind.b");
2233      else outline("lind.w");
2234      }
2235
2236      /*********************************/
2237      /*      Call subroutine          */
2238      /*********************************/
2239
2240  call(sname)
2241      char *sname;
2242      {
2243      outtab("call");
2244      tab; outstr("_"); outstr(sname);
```

```
2245      nl;
2246      }
2247
2248      /*********************************/
2249      /*  Subroutine call to address  */
2250      /*   on top of stack, return    */
2251      /*    address left on stack     */
2252      /*********************************/
2253
2254  callstk()
2255      {
2256      outline("scall");
2257      sp=sp+intwidth;
2258      }
2259
2260      /*********************************/
2261      /*  Jump to specified internal  */
2262      /*         label number         */
2263      /*********************************/
2264
2265  jump(label)
2266      int label;
2267      {
2268      outtab("ujump");
2269      tab; outlabel(label);
2270      nl;
2271      }
2272
2273      /*********************************/
2274      /* Jump to specified label if   */
2275      /* primary reg  is false (zero) */
2276      /*********************************/
2277
2278  testjump(label)
2279      int label;
2280      {
2281      outtab("fjump");
2282      tab; outlabel(label);
2283      }
2284
2285      /*********************************/
2286      /* Modify stack pointer to new  */
2287      /*       value indicated        */
2288      /*********************************/
2289
2290  modstk(newsp)
2291      int newsp;
2292      {
2293      int k;
2294      k=newsp-sp;
2295      if (k==0) return newsp;
2296      outtab("modstk");
2297      tab; outdec(k);
2298      nl;
2299      return newsp;
```

```
2300     }
2301
2302     /*  start of main segment           */
2303 header()    { outline("start"); }
2304     /*  swap primary and secondary      */
2305 swap()      { outline("swap"); }
2306     /*  partial load immediate          */
2307 immed()     { outtab("limm");tab; }
2308     /*  push primary onto stack         */
2309 push()      { outline("push"); sp=sp-intwidth; }
2310     /*  pop top of stack into secondary */
2311 pop()       { outline("pop"); sp=sp+intwidth; }
2312     /*  swap primary and top of stack   */
2313 swapstk()   { outline("xchange"); }
2314     /*  return from subroutine          */
2315 ret()       { outline("return"); }
2316
2317     /*********************************/
2318     /*   Arithmetic and logical      */
2319     /*  instructions - result in     */
2320     /*      primary register         */
2321     /*********************************/
2322
2323     /*  scale primary by n                      */
2324 scale(n)
2325     int n;
2326     {
2327     outtab("scale");
2328     tab; outdec(n); nl;
2329     }
2330     /*  add primary and secondary               */
2331 add()       { outline("add"); }
2332     /*  subtract primary from secondary         */
2333 sub()       { outline("sub"); }
2334     /*  multiply primary and secondary          */
2335 mult()      { outline("mult"); }
2336     /*  divide secondary by primary             */
2337     /*  quotient in primary, rem in secondary   */
2338 div()       { outline("div"); }
2339     /*  mod of secondary divided by primary     */
2340     /*  rem in primary, quotient in secondary   */
2341 mod()       { div(); swap(); }
2342     /*  inclusive or of primary and secondary   */
2343 or()        { outline("or"); }
2344     /*  exclusive or of primary and secondary   */
2345 xor()       { outline("xor"); }
2346     /*  logical and of primary and secondary    */
2347 and()       { outline("and"); }
2348     /*  arithmetic shift right of secondary,    */
2349     /*  number of times in primary              */
2350 asr()       { outline("asr"); }
2351     /*  arithmetic shift left                   */
2352 asl()       { outline("asl"); }
2353     /*  twos complement of primary              */
2354 neg()       { outline("neg"); }
```

```
2355      /*  increment primary by n                  */
2356  inc(n)
2357      int n;
2358      {
2359      outtab("inc");
2360      tab; outdec(n); nl;
2361      }
2362      /*  decrement primary by n                  */
2363  dec(n)
2364      int n;
2365      {
2366      outtab("dec");
2367      tab; outdec(n); nl;
2368      }
2369
2370      /*********************************/
2371      /* Conditional instructions -    */
2372      /* compare secondary against     */
2373      /* primary, put 1 in primary if */
2374      /*      true, otherwise 0        */
2375      /*********************************/
2376
2377      /*  test for =                  */
2378  eq()        { outline("testeq"); }
2379      /*  !=                          */
2380  ne()        { outline("testne"); }
2381      /*  < (signed)                  */
2382  lt()        { outline("testlt"); }
2383      /*  <=                          */
2384  le()        { outline("testle"); }
2385      /*  > (signed)                  */
2386  gt()        { outline("testgt"); }
2387      /*  >= (signed)                 */
2388  ge()        { outline("testge"); }
2389      /*  <  (unsigned)               */
2390  ult()       { outline("testult"); }
2391      /*  <= (unsigned)               */
2392  ule()       { outline("testule"); }
2393      /*  > (unsigned)                */
2394  ugt()       { outline("testugt"); }
2395      /*  >= (unsigned)               */
2396  uge()       { outline("testuge"); }
2397
2398      /*********************************/
2399      /*    Dump the literal pool      */
2400      /*********************************/
2401
2402  dumplits()
2403      {
2404      int j,k;
2405
2406      if (litptr==0) return;     /* if nothing there, exit  */
2407      outlabel(litlab); colon;   /* print literal label     */
2408      k=0;                       /* init an index ...       */
2409      while (k<litptr)           /* ... to loop with        */
```

```
2410        {
2411        defbyte();              /* pseudo-op to define byte*/
2412        j=5;                    /* max bytes per line      */
2413        while (j--)
2414            {
2415            outdec((litq[k++]));
2416            if ((j==0) | (k>=litptr))
2417                {  nl; break; }
2418            comma;              /* separate bytes          */
2419            }
2420        }
2421    }
2422
2423    /*********************************/
2424    /*   Dump all static variables  */
2425    /*********************************/
2426
2427 dumpglbs()
2428    {
2429    int j;
2430
2431    if (glbflag==0) return;   /* don't if user said no   */
2432    cptr=startglb;
2433    while (cptr<glbptr)
2434        {
2435        if (cptr[ident]!=function)
2436                        /* do if anything but function */
2437            {
2438            outstr(cptr); colon;
2439                        /* output name as label        */
2440            defstorage(); /* define storage             */
2441            j=((cptr[offset]&255)+
2442                ((cptr[offset+1]&255)<<8));
2443                        /* calculate # bytes           */
2444            if ((cptr[type]==cint) |
2445                (cptr[ident]==pointer))
2446                j=j*intwidth;
2447            outdec(j);    /* need that many            */
2448            nl;
2449            }
2450        cptr=cptr+symsiz;
2451        }
2452    }
2453
2454    /*  literal definitions         */
2455 defbyte()   { outtab("db "); }
2456 defstorage()    { outtab("ds "); }
2457
2458    /*  end of assembly             */
2459 trailer()
2460    {
2461    outtab("end");
2462    if (hasmain) outtab("start");
2463    nl;
2464    }
```

```
2465
2466
2467      /*********************************/
2468      /*  <<< end of the compiler >>> */
2469      /*********************************/
[ style 87.8 ]
```

CROSS-REFERENCE LISTING

```
NULL                1933  1952  1965

add()                795   977 *2331
addglb()             290  1040  1113 *1167
addloc()             307  1156 *1213
addmac()             263 *1260
address             *344   364   366   367
addwhile()           459 *1309
alpha()             1352 *1634
amatch()             254   256   322   324   384   386   390   392
                     394   396   398   434 *1530
an()                1354  1503  1504  1540  1624  1624  1625 *1648
                    1785  1788
and()                615 *2347
argptr              *345   360   362   363   364   365   366   367
argstk              *161   299   307   308   308   319   320   349
                     371   371
argtop              *276   319   323   325   341  *342   364
array                #71   967  1027  1033  1110  1143
ask()                227 *1849
asl()                770 *2352
asr()                763 *2350
astreq()             294 *1489  1537

bslch                #30  1422  1429
bspch                #24  1426

c                   *345 *1368  1380  1381 *1418  1422  1424  1425
                    1426  1427  1428  1429  1430  1431  1433 *1446
                    1634 *1635  1637  1637  1638  1638  1639  1642
                   *1643  1645  1645  1648 *1649  1651  1651  1652
                    1652  1653  1653  1654 *1738  1795  1795  1799
                    1799  1837 *1838  1840  1842  2056 *2057  2059
                    2062  2065  2066 *2077  2088  2089  2089
call()              1079 *2240
callfunction()       984   988   990 *1061
callstk()           1080 *2254
cchar                #78   255   323   385  1153  2176  2203  2219
                    2232
ch                  *152   313   383   562   584   606   630   630
                     660   660   703   703   723   723   754   754
                     783   783   818   818   818   953   953  1069
                    1277  1277  1352  1354  1377  1378  1450  1452
```

	1520	1521	1521	1521	1537	1539	1539	1539
	1540	1585	1585	1606	1612	1613	1624	1625
	1626	1665	1683	1705	1707	1709	1715	1723
	1723	1741	1741	1742	1744	1744	1747	1747
	1749	1751	1753	1755	1762	1766	1768	1775
	1778	1780	1785	1788	1790	1806	1806	1810
	1811	1820	1820	1820	1829	1830	1891	1919
	1919	1932	1951	1965	2014			
charwidth	#19	861	873	918	922	933	937	1153
chmax	*152	193	1705					
cint	#79	257	290	325	387	793	803	859
	871	888	916	920	931	935	975	1026
	1040	1144	2444					
clearscreen	#20	1852						
closeout()	237	*1974	2063					
cmode	*173	221	513	523	1717	1740		
colon	#37	296	436	440	444	460	464	2407
	2438							
comma	#38	2418						
comment	#40	230	1594	1719	1993	2012	2012	2020
compound()	389	*411						
constant()	1045	*1392						
cptr	*178	670	671	676	677	689	690	695
	696	709	710	715	716	729	730	735
	736	791	792	793	801	802	803	1173
	1173	1179	1181	1182	1183	1184	1185	1187
	1219	1219	1225	1227	1228	1229	1230	1231
	1233	2432	2433	2435	2438	2441	2442	2444
	2445	2450	2450					
crch	#28	1427						
ctext	*172	1717	1868	1870				
dec()	872	873	921	922	932	933	*2363	
declglb()	255	257	*1092					
declloc()	385	387	*1124					
defaults()	1863	*1875						
defbyte()	2411	*2455						
defstorage()	2440	*2456						
delwhile()	467	*1321						
display()	1853	1854	1857	1858	1859	1860	1862	1869
	1882	1889	1897	*1905	1930	1949	1953	2023
	2023							
div()	834	*2338	2341					
doasm()	259	402	*511					
dobreak()	357	*485						
docont()	399	*498						
doif()	391	*423						
doreturn()	395	*474						
dowhile()	393	*451						
dumpglbs()	235	*2427						
dumplits()	234	*2402						
endglb	#49	#50	1174					
endloc	#51	1220						
eof	*166	199	252	383	415	518	1610	1667

```
                1684  1699  1700  1781  1948  1951
eolch            #26   #35  1424  1705  1911
eq()             639 *2378
errcnt          *163   207  2025  2037
errlval()        546   855   867   912   927 *1984
errmulti()       286   287   304  1106  1137 *1989
errname()        351  1105  1136  1266 *1997
error()          280   293   311   315   326   369   373   896
                 960   969  1048  1176  1222  1276  1281  1315
                1329  1455  1557  1562  1575  1593  1756  1769
                1805  1934  1966  1986  1992  1999 *2007  2034
                2063
errorsummary()   238 *2032
expression()     403   476  *531   973  1072  2140

fclose()        1712  1976
ffch             #27  1428
findglb()        284  1029  1106  1173 *1194
findloc()        304   360  1021  1137  1219 *1240
findmac()       *1289  1794
flab1           *425   429   430   436   440
flab2           *425   439   444
flev            *425   427   433   443
fopen()         1933  1952  1965
fsp             *425   428   432   442
function         #73   286   287   288   290   290   985   996
                1029  1040  2435

gch()           1277  1278  1354  1422  1424  1456  1459  1461
                1612  1613  1624  1626  1671  1685  1747  1752
                1757  1759  1765  1770  1772  1791  1802 *1827
ge()             701 *2388
getarg()         323   325  *341
getc()          1703
getlabel()       429   439   457   458 *1337  1865
getloc()        1023 *2187
getmem()        2164 *2173
getqchar()      1396 *1414
getqstring()    1397 *1443
gets()          1890  1918  1931  1950
glbflag         *170  1856  1883  2431
glbptr          *136   191  1174  1179  1186  1186  1200  2433
gt()             741 *2386

hasmain         *179   210   295  2462
header()         295 *2303
hier1()          535  *538   548  1015
hier10()         816   832   839  *847   855   867   879   886
                 895
hier11()         909  *944
hier2()          543  *555
hier3()          560   569  *577
hier4()          582   591  *599
hier5()          604   613  *621
hier6()          626   637   644  *652
```

hier7() 657 668 687 707 727 *747
hier8() 752 761 768 *776
hier9() 781 790 800 *811 825

id 1167 *1168 1181 1213 *1214 1227
ident #57 286 362 671 677 690 696 710
716 730 736 792 802 859 871 916
920 931 935 965 967 985 996 1026
1027 1029 1033 1181 1227 2176 2203 2435
2445
immed() 900 998 1034 1049 1395 1396 1399 *2307
inbyte() 354 705 725 1380 1540 1624 *1663
inc() 860 861 917 918 936 937 *2356
inchar() *1681 1777 1777 1780 1783 1783
indirect() 2165 *2229
inline() 516 1608 1668 1683 *1692 1780
input *167 200 1698 1701 1713 1944 1945 1952
1953
input2 *169 201 1701 1713 1713 1965 1966
inscnt *164 208 2042 2130
intwidth #18 308 371 793 803 860 872 917
921 932 936 975 1075 1144 1149 1154
2257 2309 2311 2446

j *344 350 350 356 362 *1095 1103 1104
1110 1111 1113 *1127 1134 1135 1143 1148
1153 1156 *2404 2412 2413 2416 *2429 2441
2446 2446 2447
jump() 439 463 491 504 *2265

k *541 543 546 552 *558 560 562 563
*580 582 584 585 *602 604 606 607
*624 626 629 630 631 *655 657 660
661 662 *750 752 754 755 *779 781
783 784 *814 816 818 819 *850 855
867 879 880 886 887 895 896 909
912 927 940 *947 950 970 980 991
993 995 1003 *1011 1015 1017 *1095 1102
1109 1110 1113 *1127 1140 1141 1144 1144
1149 1153 1154 1155 *1312 1316 1317 1318
*1349 1353 1354 1355 1356 *1367 1370 1371
1373 1374 1375 1381 1381 1383 1383 1384
*1417 1420 1424 1425 1426 1427 1428 1429
1430 1431 1433 1435 *1473 1475 1479 1481
*1493 1495 1496 1501 1505 *1517 1520 1521
*1534 1537 1539 *1694 1703 1705 1707 1710
*1737 1787 1790 1790 1793 1793 1798 1799
*1877 1892 *1908 1910 1912 1912 *2010 2013
2014 2016 2017 *2076 2080 2086 2088 2089
2090 2091 2091 *2102 2105 2105 2106 *2293
2294 2295 2297 *2404 2408 2409 2415 2416
keepch() 1746 1751 1757 1760 1764 1770 1773 1795
1799 1802 1804 *1837

label 2113 *2114 2117 2136 *2137 2142 2265 *2266

```
                2269   2278 *2279   2282
laststst        *175    204    391    393    395    397    399    402
                 403    404
le()             682  *2384
legalname       *344    351    358
len             1489  *1491   1496   1530  *1532   1537
line            *150    193    519   1715   1720   1723   1741   1806
                1811   1820   1890   1918   1931   1933   1950   1952
                2012   2013   2023
linemax         #112    193    194
linesize        #111   #112   *150   *151
lit             1514  *1515   1520   1530  *1531   1537
litabsz         #105   #106   *140
litlab          *159   1400   1865   2407
litmax          #106   1453
litptr          *141    197   1449   1453   1459   1462   2406   2409
                2416
litq            *140   1459   1462   2415
lncnt           *165    209   1706   2047
locptr          *136    192    298    334    427    433    443    455
                 465   1220   1225   1232   1232   1246
lt()             721  *2382
lval            *533    535    535    538   *539    543    547    549
                 555   *556    560    563    577   *578    582    585
                 599   *600    604    607    621   *622    626    631
                 652   *653    657    662    670    689    709    729
                 747   *748    752    755    776   *777    781    784
                 791    801    811   *812    816    819    847   *848
                 855    856    857    858    862    867    868    869
                 870    874    879    880    886    887    888    889
                 889    890    895    897    901    903    909    913
                 914    915    919    928    929    930    934    944
                *945    950    951    966    978    979    987    991
                1006  *1007   1015   1024   1025   1026   1031   1032
                1037   1041   1042   1045   1045   2150  *2151   2153
                2153   2154   2161  *2162   2164   2164   2164   2165
lval2           *541    548    548   *558    569    569   *580    591
                 591   *602    613    613   *624    637    637    644
                 644   *655    668    668    676    687    687    695
                 707    707    715    727    727    735   *750    761
                 761    768    768   *779    790    790    800    800
                *814    825    825    832    832    839    839

macb            *143    214    217
macbmax         *146    217   1280
macbptr         *145    214   1274   1278   1280
macbsize        #117   *143    217
macp            *147    215    216   1295
macpmax         *148    216   1271
macpptr         *148    215   1271   1274   1296
macssize        #119   *144
mact            *144    213   1295
mactptr         *145    213   1270   1274   1274
mactsize        #118   *147    216
main()          *189
```

match() 258 260 262 292 300 315 350 352
373 388 400 401 414 517 544 566
588 610 634 641 665 684 758 765
787 797 822 829 836 853 865 877
884 893 910 925 956 982 1013 1076
1103 1107 1116 1134 1138 1159 1374 1375
1421 1434 1448 1456 *1514 1554 1575 1591
minus *1367 1370 1375 1375 1383
mline *151 194 1741 1806
mn *1262 1270 1273
mod() 841 *2341
modstk() 330 432 442 466 477 490 503 1081
1155 *2290
mpmax *153 194 1805 1841
mptr *152 1741 1805 1806 1810 1840 1841 1841
mq *1738 1794 1795
mqp *1292 1295 1296 1299 1300
mqt *1293 1295 1298 1299 1300 1300
mulfile *171 1699 1856 1898
mult() 827 *2335

n *275 278 284 286 287 290 294 296
302 304 304 307 *345 351 360 2324
*2325 2328 2356 *2357 2360 2363 *2364 2367
name #56 2179 2206
namemax #65 1790
namesize #64 *275 *345 *1009 *1095 *1127 *1262 1274
1300 *1738
nargs *1064 1066 1075 1075 1081
ncmp *162 203 413 416 2034
ne() 646 *2380
needbrack() 962 974 1016 1078 1565 *1588 2139 2141
needsemi() 255 257 323 325 385 387 395 397
399 403 *1573
needstend() 317 355 372 476 1071 1101 1133 *1582
needsub() 1109 1140 *1550
neg() 881 *2354
newfunc() 264 *273
newsp 2290 *2291 2294 2295 2299
nexch *152 629 661 661 1521 1539 1723 1741
1775 1778 1806 1820 1830
nl #35 232 296 436 440 444 460 464
520 902 1000 1036 1051 1405 1594 1721
1855 1855 1861 1994 2012 2020 2024 2036
2039 2041 2044 2046 2049 2126 2143 2192
2207 2245 2270 2298 2328 2360 2367 2417
2448 2463
num *1010 1045 *1552 1555 1558 1560 1563 1563
1566 *1877 1891 1892 1894 2073 *2074 2081
2083 2083 2088 2090 2090
number() *1364 1395 1555 1892
numeric() 1377 1378 *1642
numglbs #47 #49
nxtlab *158 1339 1856 1894

```
offset              #60   287   288   364   365   366   367  1184
                   1185  1230  1231  2191  2191  2441  2442
openin()            229  1699 *1942
openinclude()       261 *1962
openout()           228 *1926
or()                571 *2343
outbyte()           #35   #36   #37   #38   #39   #40  1401  2019
                   *2056  2084  2089  2106
outdec()           1050  1404  2037  2042  2047 *2073  2117  2191
                   2297  2328  2360  2367  2415  2447
outlabel()          436   440   444   460   464  1400 *2113  2269
                   2282  2407
outline()          *2124  2179  2219  2220  2232  2233  2256  2303
                   2305  2309  2311  2313  2315  2331  2333  2335
                   2338  2343  2345  2347  2350  2352  2354  2378
                   2380  2382  2384  2386  2388  2390  2392  2394
                   2396
output             *168   202  1929  1933  1934  1976  1976  1977
                   2060  2062
outstr()            231   296   296   519   901   999  1035  1594
                   1720  1994  2012  2020  2021  2022  2038  2043
                   2048 *2099  2116  2130  2206  2244  2244  2438
outtab()           2126 *2128  2177  2178  2190  2204  2205  2243
                   2268  2281  2296  2307  2327  2359  2366  2455
                   2456  2461  2462

parse()             233  *250
pointer             #72   350   356   671   677   690   696   710
                    716   730   736   792   802   859   871   916
                    920   931   935   965  1026  1103  1111  1134
                   1148  1153  2176  2203  2445
pop()               570   592   614   638   645   669   688   708
                    728   762   769   794   804   826   833   840
                    976  2218 *2311
preprocess()       1609  1669 *1735
primary()           950 *1006
ptr                *275   284   286   287   288  *487   489   490
                    491  *500   502   503   504  *851   858   859
                    859   870   871   871   889   889   901   903
                    915   916   916   920   920   930   931   931
                    935   935  *948   951   958   965   967   975
                    979   984   985   990   995   996   999 *1009
                   1021  1023  1024  1025  1026  1027  1029  1029
                   1031  1033  1035  1037  1040  1041  1061 *1062
                   1068  1073  1079  1079 *1171  1179  1180 *1197
                   1199  1200  1202  1203  1203  1204  1204 *1217
                   1225  1226 *1243  1245  1246  1248  1249  1249
                   1250  1250  1309 *1310  1318  2007 *2008  2021
                   2023  2099 *2100  2105  2106  2124 *2125  2126
                   2128 *2129  2130
push()              547   568   590   612   636   643   667   686
                    706   726   760   767   789   799   824   831
                    838   856   868   913   928   972  1068  1074
                   *2309
putc()             2062
```

putchar() #20 #20 #20 1911 1912 2065
putmem() 2153 *2200
putstk() 2154 *2215

quoch #29 1762 1764 1766 1773
quote *177 219 220 1448 1456

readwhile() 489 502 1323 *1326
reply() 1863 1870 1883 1898 *1915
resetptr() 281 522 1267 1702 *1818 1851 1872 1917
1928 1935 1947 1955 1967
ret() 331 478 *2315
rvalue() 535 548 563 569 585 591 607 613
631 637 644 662 668 687 707 727
755 761 768 784 790 800 819 825
832 839 857 869 880 887 914 929
966 987 *2161

scale() 793 803 975 *2324
skipblanks() 265 312 561 583 605 627 658 753
782 817 952 1067 1351 1519 1536 1584
*1602 1627 1964
skipchars() 311 961 1052 *1622 1999
sn *1262 1270 1273
sname *1009 1019 1021 1029 1040 *1095 1105 1106
1106 1113 *1127 1136 1137 1137 1156 1167
*1168 1173 1180 1194 *1195 1202 1203 1213
*1214 1219 1226 1240 *1241 1248 1249 *1262
1264 1270 1289 *1290 1298 1299 1346 *1347
1354 1355 1356 *1738 1790 1793 1793 1794
1799 1989 *1990 1994 2240 *2241 2244
sp *160 198 319 333 428 432 442 456
466 1081 1081 1155 1155 1156 2191 2257
2257 2294 2309 2309 2311 2311
space #39 2016
startglb #48 #49 191 1199 2432
startloc #50 192 298 334 1245
stasm #129 402
statement() 328 *381 415 431 441 462
statik #84 1183
stbreak #127 397
stcont #128 399
stexp #130 403
stif #124 391
stkloc #85 1229
storage #59 1183 1229
store() 549 862 874 919 934 *2150
str 1588 *1589 1591 1594 1905 *1906 1912 1912
str1 1470 *1471 1478 1479 1489 *1490 1498 1499
1501 1503
str2 1470 *1471 1476 1478 1479 1489 *1490 1498
1500 1501 1504
streq() 313 703 703 723 723 754 754 1203
1249 1299 *1470 1520 1585
streturn #126 328 395

```
stwhile          #125   393
sub()             805 *2333
swap()           *2305  2341
swapstk()        1073 *2313
sym              2173 *2174  2176  2176  2179  2187 *2188  2191
                 2191  2200 *2201  2203  2203  2206
symname()         278   302   351  1019  1105  1136  1264 *1346
symsiz            #45   #49   #50   #51  1186  1204  1232  1250
                 2450
symtab            #48   #51  *135
symtbsz           #46   #51  *135


t                 341  *342   363
tab               #36  2016  2130  2190  2206  2244  2269  2282
                 2297  2307  2328  2360  2367
tabch             #25   #36  1277  1425  1613  1744  1747  2016
test()            430   461 *2136
testjump()       2142 *2278
trailer()         236 *2459
typ              1092 *1093  1113  1124 *1125  1144  1153  1156
                 1167 *1168  1182  1213 *1214  1228
type              #58   363   793   803   859   871   889   903
                  916   920   931   935   975   979  1025  1037
                 1182  1228  2176  2203  2444
typeobj          2215 *2216  2219  2229 *2230  2232


uge()             692   698 *2396
ugt()             732   738 *2394
ule()             673   679 *2392
ult()             712   718 *2390
unit             *1694  1701  1701  1703  1712


val              1364 *1365  1384  1392 *1393  1395  1396  1397
                 1404  1414 *1415  1435  1443 *1444  1449
value            1167 *1169  1184  1185  1213 *1215  1230  1231
variable          #70   350  1104  1135


wq                #92  *137   195  *453   455   456   457   458
                  459   460   461   463   464   465   466  1328
wqlab            #100   458   461   464   491
wqloop            #99   457   460   463   504
wqmax             #92  1314
wqptr            *138   195  1314  1318  1323  1323  1328  1330
wqsiz             #91   #92  1317  1323  1330
wqsp              #98   456   466   490   503
wqsym             #97   455   465
wqtabsz           #90   #92  *137


xor()             593 *2345


zs               *2076  2079  2089  2089
```

FUNCTION LIST (SORTED BY LINE NUMBER)

```
 189  main()
 250  parse()
 273  newfunc()
 341  getarg(t,argto
 381  statement()
 411  compound()
 423  doif()
 451  dowhile()
 474  doreturn()
 485  dobreak()
 498  docont()
 511  doasm()
 531  expression()
 538  hier1(lval)
 555  hier2(lval)
 577  hier3(lval)
 599  hier4(lval)
 621  hier5(lval)
 652  hier6(lval)
 747  hier7(lval)
 776  hier8(lval)
 811  hier9(lval)
 847  hier10(lval)
 944  hier11(lval)
1006  primary(lval)
1061  callfunction(p
1092  declglb(typ)
1124  declloc(typ)
1167  addglb(sname,i
1194  findglb(sname)
1213  addloc(sname,i
1240  findloc(sname)
1260  addmac()
1289  findmac(sname)
1309  addwhile(ptr)
1321  delwhile()
1326  readwhile()
1337  getlabel()
1346  symname(sname)
1364  number(val)
1392  constant(val)
1414  getqchar(val)
1443  getqstring(val
1470  streq(str1,
1489  astreq(str1,
1514  match(lit)
1530  amatch(lit,
1550  needsub()
1573  needsemi()
1582  needstend()
1588  needbrack(str)
1602  skipblanks()
1622  skipchars()
1634  alpha(c)
1642  numeric(c)
1648  an(c)
1663  inbyte()
1681  inchar()
1692  inline()
1735  preprocess()
1818  resetptr()
1827  gch()
1837  keepch(c)
1849  ask()
1875  defaults()
1905  display(str)
1915  reply()
1926  openout()
1942  openin()
1962  openinclude()
1974  closeout()
1984  errlval()
1989  errmulti(sname
1997  errname()
2007  error(ptr)
2032  errorsummary()
2056  outbyte(c)
2073  outdec(num)
2099  outstr(ptr)
2113  outlabel(label
2124  outline(ptr)
2128  outtab(ptr)
2136  test(label)
2150  store(lval)
2161  rvalue(lval)
2173  getmem(sym)
2187  getloc(sym)
2200  putmem(sym)
2215  putstk(typeobj
2229  indirect(typeo
2240  call(sname)
2254  callstk()
2265  jump(label)
2278  testjump(label
2290  modstk(newsp)
2303  header()
2305  swap()
2307  immed()
2309  push()
2311  pop()
2313  swapstk()
2315  ret()
2324  scale(n)
2331  add()
2333  sub()
2335  mult()
2338  div()
2341  mod()
2343  or()
2345  xor()
2347  and()
2350  asr()
2352  asl()
2354  neg()
2356  inc(n)
2363  dec(n)
2378  eq()
2380  ne()
2382  lt()
2384  le()
2386  gt()
2388  ge()
2390  ult()
2392  ule()
2394  ugt()
2396  uge()
2402  dumplits()
2427  dumpglbs()
2455  defbyte()
2456  defstorage()
2459  trailer()
```

FUNCTION LIST (SORTED BY NAME)

2331 add()
1167 addglb(sname,i
1213 addloc(sname,i
1260 addmac()
1309 addwhile(ptr)
1634 alpha(c)
1530 amatch(lit,
1648 an(c)
2347 and()
1849 ask()
2352 asl()
2350 asr()
1489 astreq(str1,
2240 call(sname)
1061 callfunction(p
2254 callstk()
1974 closeout()
411 compound()
1392 constant(val)
2363 dec(n)
1092 declglb(typ)
1124 declloc(typ)
1875 defaults()
2455 defbyte()
2456 defstorage()
1321 delwhile()
1905 display(str)
2338 div()
511 doasm()
485 dobreak()
498 docont()
423 doif()
474 doreturn()
451 dowhile()
2427 dumpglbs()
2402 dumplits()
2378 eq()
1984 errlval()
1989 errmulti(sname
1997 errname()
2007 error(ptr)
2032 errorsummary()
531 expression()
1194 findglb(sname)
1240 findloc(sname)
1289 findmac(sname)
1827 gch()
2388 ge()
341 getarg(t,argto
1337 getlabel()
2187 getloc(sym)
2173 getmem(sym)
1414 getqchar(val)
1443 getqstring(val
2386 gt()
2303 header()
538 hier1(lval)
847 hier10(lval)
944 hier11(lval)
555 hier2(lval)
577 hier3(lval)
599 hier4(lval)
621 hier5(lval)
652 hier6(lval)
747 hier7(lval)
776 hier8(lval)
811 hier9(lval)
2307 immed()
1663 inbyte()
2356 inc(n)
1681 inchar()
2229 indirect(typeo
1692 inline()
2265 jump(label)
1837 keepch(c)
2384 le()
2382 lt()
189 main()
1514 match(lit)
2341 mod()
2290 modstk(newsp)
2335 mult()
2380 ne()
1588 needbrack(str)
1573 needsemi()
1582 needstend()
1550 needsub()
2354 neg()
273 newfunc()
1364 number(val)
1642 numeric(c)
1942 openin()
1962 openinclude()
1926 openout()
2343 or()
2056 outbyte(c)
2073 outdec(num)
2113 outlabel(label
2124 outline(ptr)
2099 outstr(ptr)
2128 outtab(ptr)
250 parse()
2311 pop()
1735 preprocess()
1006 primary(lval)
2309 push()
2200 putmem(sym)
2215 putstk(typeobj
1326 readwhile()
1915 reply()
1818 resetptr()
2315 ret()
2161 rvalue(lval)
2324 scale(n)
1602 skipblanks()
1622 skipchars()
381 statement()
2150 store(lval)
1470 streq(str1,
2333 sub()
2305 swap()
2313 swapstk()
1346 symname(sname)
2136 test(label)
2278 testjump(label
2459 trailer()
2396 uge()
2394 ugt()
2392 ule()
2390 ult()
2345 xor()

Appendix 3: C Style Analysis

The features of a program that contribute to its 'elegance' are very much subjective, and often instinctive. A superficial analysis of a program's 'style' (that is, its visual presentation), while not being the only factor, is certainly an indicative, and easily automated, component.

Presented here is a suite of programs (specifically using some of the many 'software tools' available under the UNIX operating system, but generally programmable in any high-level language) that performs a textual analysis of a C program, yielding a percentage 'style score'.

STYLE ANALYSIS

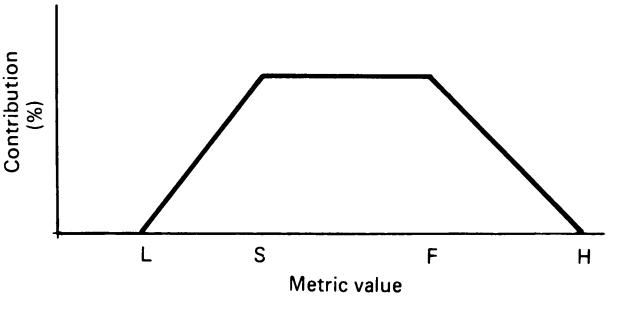
The features that contribute to the style score are based on proposals made by Rees (1982), adapted for C rather than Pascal:

Module length	The average length, in non-blank lines, of function definitions; functions that are prolific and too short tend to obscure the program logic, while those that are too long are difficult to dismember.
Identifier length	The average length, in characters, of user identifiers; brief identifier names (such as i or c) are often meaningless, while overlong names make the program verbose (most programmers will know that selection of pithy, meaningful identifier names is often one of the most time-consuming operations in writing a program).
Comments	The percentage of all lines that contain comments; over-commenting is as much of a sin as under-commenting; some comments, however, are always necessary, even in the shortest of programs.
Indentation	The ratio of initial spaces to total number of characters; indentation can be used to good effect to indicate the program structure.

Blank lines	The percentage of all lines that are blank; blank lines separate functional units of a program.
Line length	The average number of non-blank characters per line; sensible use of multiple statement lines can make a program visually concise, but not obscure.
Embedded spaces	The average number of embedded spaces per line; embedded spaces do for a line what blank lines do for a function.
Constant definitions	The percentage of all user identifiers that are defined constants: use of manifest constants not only makes a program easier to modify, it also associates meaning with the constant.
Reserved words	The number of different reserved words and standard functions used; the variety of reserved words used is indicative of command of the language.
Included files	The extent to which a program is segmented by using "#include" files; separating functional units of a program into different files breaks it down into more manageable chunks.
Goto statements	The number of occurrences of a "goto" statement; advocates of structured programming will usually allow the use of a single "goto" in a program to handle a special exit condition – more than that is a cardinal sin.

A score is associated with each of the above metrics, each contributing a different maximum percentage to the final score, in recognition of the fact that some factors are more important than others. All scores are additive, with the exception of the last, which is subtractive. Too high or too low a figure for each metric is detrimental to the final score.

The individual score is determined by reference to a table which specifies, for each metric (see figure below)



(1) the point L, below which no score is obtained
(2) the point S, the start of the 'ideal' range for the metric

(3) the point F, the finish of the ideal range
(4) the point H, above which no score is obtained.

Values between S and F score maximum marks; those between L and S, and F and H, score marks depending on their exact position within the range.

THE STYLE COMMAND

This is a command file, written in the UNIX command language (the 'shell'), to control the application of the various programs within the suite to the data (that is, the C program being analysed).

```
: ----- initialise some variables
TMP1=/tmp/TMP1$$
TMP2=/tmp/TMP2$$
RESULTS=/tmp/STATS$$
LIB=.

: ----- clean up on exit
trap "rm -f $TMP1 $TMP2 $RESULTS; trap ''0; exit" 0 1 2 13 15

: ----- analyse all programs presented
for i do
    echo; echo 'Style analysis of' $i
    if test -r $i
    then
                    : ----- count comment lines and total lines
        awk -f $LIB/style.cnt.awk $i >$RESULTS

                    : ----- replace tabs by spaces
                    : ----- convert to lower case
                    : ----- remove strings
                    : ----- remove comments
        $LIB/style.detab < $i |\
            tr 'A-Z' 'a-z' |\
            awk "/^main[ ]*\(/ { flag=1 }\
                                { if (flag+0) print }" |\
            sed -f $LIB/style.str.sed |\
            sed -n -f $LIB/style.com.sed >$TMP1
```

```
                    : ----- sort program words
        tr -cs 'a-z0-9' '\012' <$TMP1 |\
            sed -n '/^[a-z]/p' |\
            sort -u >$TMP2
                    :
                    : ----- find length of user identifiers
        comm -23 $TMP2 $LIB/style.dict |\
            awk '{totl+=length};\
                  END {printf "NL ";if (NR) print (totl/NR);else print 0;
                       print "ID " NR }' >>$RESULTS

                    : ----- count variety of reserved words
        comm -12 $TMP2 $LIB/style.dict |\
            (echo -n "RW ";wc -l;) >>$RESULTS

                    : ----- produce remaining metrics
        awk -f $LIB/style.met.awk $TMP1 >>$RESULTS

                    : ----- and analyse
        $LIB/style.stan <$RESULTS
    else echo "   Cannot read"; echo
    fi
done
```

THE PROGRAM STYLE.CNT.AWK

This program uses the awk pattern processor available under UNIX to count the number of commented lines, preprocessor directive lines and the total number of lines. As with all the remaining programs in the suite, it could easily, if not so concisely, be written in C.

```
# count commented lines
{if (index($0,"/*")||index($0,"*/")) comments++}

# count preprocessor lines
/^#include/     { includes++ }
/^#define/      { defines++ }
```

```
# report include files, defines, comment lines and total lines
END     {print "IF "(includes+0); print "DF "(defines+0)
         print "CL "(comments+0); print "TL "(NR+0)}
```

THE PROGRAM STYLE. DETAB

This program is translated from the version given by Kernighan and Plauger (1976) using the Ratfor programming language.

```
/*************************************************************/
/* Detab - convert tabs to appropriate number of spaces      */
/*                                                           */
/* transcribed from Kernighan & Plauger's "Software Tools" */
/*************************************************************/

#include        <stdio.h>
#define         MAXLINE 132
#define         TABSIZE 8

main()
{       int     ch, tabs[MAXLINE], col=1;

        settabs(tabs);

        while ( (ch=getchar()) != EOF )
                if ( ch=='\t' )
                        do      { putchar(' '); col++; }
                        while   ( !tabpos(col,tabs) );
                else if ( ch=='\n' )
                        { putchar('\n'); col=1; }
                else    { putchar(ch); col++; }
}

        /* set up tab positions */
settabs(tabs)
int     tabs[MAXLINE];

{       int     i;

        for ( i=1; i<=MAXLINE; i++ )
                if ( (i%TABSIZE)==1 ) tabs[i]=1;  else tabs[i]=0;
```

```
}

        /* see if we're at a tab position */

tabpos(col, tabs)
int     col, tabs[MAXLINE];

{
        if ( col>MAXLINE ) return(1); else return(tabs[col]);
}


[ style 82.7 ]
```

THE PROGRAM STYLE.STR.SED

This program uses another of the UNIX software tools, sed, a stream editor, to remove characters between double or single quotes, to obviate their inclusion in subsequent metric calculations.

```
# Destring a C program and replace comment delimiters
# with single characters (easier later)

# take out both types of string
s/'[^']*'//g
s/"[^"]*"//g

# replace comment delimiters
s/\/\*/\/g
s/\*\//`/g
```

THE PROGRAM STYLE.COM.SED

The sed utility is used again to remove all comments now that they have been counted.

```
# Decomment a C program

:start
                                        # strip trailing spaces
/[ ]*$/         s/[ ]*$//
                                        # lose comment-only lines
```

```
/^[ ]*\.*`$/    d
                                        # strip short comments
/\.*`/          s/\[^\]*`//g
                                        # strip multi-line comments
/^[ ]*\.*$/     bloop
/\.*$/          {               s/[ ]*\.*$//p
                                :loop
                                n
                                        # ensure flag reset
                                tdummy
                                :dummy
                                        # keep going until delimiter
                                s/^[^\]*`//
                  /^[ ]*$/      d
                                tstart
                                bloop
                }
                                        # print whatever's left
                p
```

THE FILE STYLE.DICT

This file contains all words that are considered to be either reserved words or standard library functions. This can be used as a reference against which to compare all program words and hence count the reserved word usage.

```
alloc       else        for         null        stderr
argc        entry       fprintf     printf      stdin
argv        eof         fputs       putc        stdout
auto        extern      freopen     putchar     struct
break       fclose      fscanf      register    switch
case        fdopen      ftell       return      typedef
char        feof        getc        scanf       union
close       ferror      getchar     short       unsigned
continue    fgets       goto        sizeof      while
default     file        if          sprintf
do          float       int         sscanf
double      fopen       long        static
```

THE PROGRAM STYLE.MET.AWK

This program, the last stage before analysis, uses awk again to produce all the remaining metrics.

```
# Produce "style" metrics for a C program

# Compute number of blank lines
                    { if (NF==0) {blank++; next } }

# Compute number of non-blank characters and imbedded spaces
                    { nbchars=0
                      for (i=NF; i>0; i-- ) nbchars+=length($i)
                      nonblank+=nbchars
                      start=index($0,$1)
                      imbedded+=length-nbchars-(start-1) }

# Compute amount of indentation
/^[ ]/              { indented+=(index($0,$1)-1) }

# Compute total number of characters
                    { chars+=length }

# Compute number of modules
/^[a-z_][a-z_0-9]*[ a-z_0-9]*\(.*\)/     { module++ }

# Compute number of gotos
/^goto[ ]+|[ ]+goto[ ]+/                 { jumps++ }

# Report results
END {
        print "NR " (NR+0)
        print "LC " (NR-blank)
        print "NB " (nonblank+0)
        print "IN " (indented+0)
        print "TC " (chars+0)
        print "BL " (blank+0)
        print "IM " (imbedded+0)
        print "MO " (module+0)
        print "JU " (jumps+0)
```

THE PROGRAM STYLE.STAN

This program analyses the results produced by the preceding programs. The input to the program consists of a sequence of lines containing two fields each, where the first is an identifying label, and the second is the associated metric. The array 'max' represents the percentage weighting for each metric; the arrays 'lo', 'lotol', 'hitol' and 'hi' represent the points L, S, F and H respectively, as discussed earlier. It is these arrays that may need customising to reflect individual preference.

```
main()          /* analyse style results */
{
        static int
           max[] =      {  9, 12, 12, 11,  8, 15,  6, 14,-20,  5,  8 },

                      /* ch  cl  in  bl  sp  ml  rw  id  go  if  df */

           lo[] =       {  8,  8,  8,  8,  1,  4,  4,  4,  1,  0, 10 },
           lotol[] =    { 12, 15, 24, 15,  4, 10, 16,  5,  3,  3, 15 },
           hitol[] =    { 25, 25, 48, 30, 10, 25, 30, 10,199,  3, 25 },
           hi[] =       { 30, 35, 60, 35, 12, 35, 36, 14,200,  4, 30 };

        float   param[11];

        static char
           *ident[] =  { "  characters per line ",      /* ch */
                         "% comment lines       ",      /* cl */
                         "% indentation         ",      /* in */
                         "% blank lines         ",      /* bl */
                         "  spaces per line     ",      /* sp */
                            module length       ",      /* ml */
                            reserved words      ",      /* rw */
                            identifier length   ",      /* id */
                            gotos               ",      /* go */
                            include files       ",      /* if */
                         "% defines             " };    /* df */
        int     i;

        float   blank, nonblank, comments, includes, defines,
                indented, imbedded, modules, jumps, ids,
                nameleng, score, oldscore, fact, totalchars,
                wordcount, linecount, totallines, lines, f;
```

```
    char    s[8];

    for (i=0; i<16; i++)
        { scanf("%s %f",s,&f);
          if (!strcmp(s,"IF")) includes=f;
          else if (!strcmp(s,"DF")) defines=f;
          else if (!strcmp(s,"NR")) lines=f;
          else if (!strcmp(s,"NL")) nameleng=f;
          else if (!strcmp(s,"ID")) ids=f;
          else if (!strcmp(s,"RW")) wordcount=f;
          else if (!strcmp(s,"CL")) comments=f;
          else if (!strcmp(s,"TL")) totallines=f;
          else if (!strcmp(s,"LC")) linecount=f;
          else if (!strcmp(s,"NB")) nonblank=f;
          else if (!strcmp(s,"IN")) indented=f;
          else if (!strcmp(s,"TC")) totalchars=f;
          else if (!strcmp(s,"BL")) blank=f;
          else if (!strcmp(s,"IM")) imbedded=f;
          else if (!strcmp(s,"MO")) modules=f;
          else if (!strcmp(s,"JU")) jumps=f; }

printf("\n    TC    TL    MO    LC    BL    CL    NB    IN    RW    ID");
printf("    IM    NL    IF    DF    JU\n");

    /* total characters, excluding comment-only lines */
printf("%6ld", (long int)totalchars);
    /* total lines */
printf("%5d", (int)totallines);
    /* number of function definitions */
printf("%5d", (int)modules);
    /* number of lines, excluding comment-only lines & blank lines */
printf("%5d", (int)linecount);
    /* number of blank lines */
printf("%5d", (int)blank);
    /* number of lines containing comments */
printf("%5d", (int)comments);
    /* number of non-blank characters, excluding comment-only lines */
printf("%6ld", (long int)nonblank);
```

```
    /* number of leading spaces (amount of indentation) */
printf("%6ld", (long int)indented);
    /* number of different reserved words */
printf("%5d", (int)wordcount);
    /* number of user identifiers */
printf("%5d", (int)ids);
    /* number of embedded spaces, excluding comments */
printf("%5d", (int)imbedded);
    /* average length of user identifiers */
printf("%5.2f", nameleng);
    /* number of #include's */
printf("%5d", (int)includes);
    /* number of #define's */
printf("%5d", (int)defines);
    /* number of goto's */
printf("%5d\n", (int)jumps);

                    linecount=lines-blank;
    if (linecount)  nonblank/=linecount;
    if (totallines) comments/=(totallines/100);
    if (totalchars) indented/=(totalchars/100);
    if (lines)      blank/=(lines/100);
    if (linecount)  imbedded/=linecount;
    if (modules)    modules=linecount/modules;
    if (ids)        defines/= (ids/100);

    param[0]=nonblank;
    param[1]=comments;
    param[2]=indented;
    param[3]=blank;
    param[4]=imbedded;
    param[5]=modules;
    param[6]=wordcount;
    param[7]=nameleng;
    param[8]=jumps;
    param[9]=includes;
    param[10]=defines;
```

```
        oldscore=0;

        for (i=0; i<=10; i++) {
                if (lotol[i]<=param[i] && param[i]<=hitol[i])
                        score+=max[i];
                else if (lo[i]<=param[i] && param[i]<lotol[i])
                      { fact=((param[i]-lo[i])/(lotol[i]-lo[i]));
                        score+=max[i]*fact; }
                else if (hitol[i]<param[i] && param[i]<=hi[i])
                      { fact=((hi[i]-param[i])/(hi[i]-hitol[i]));
                        score+=max[i]*fact; }
                printf("\n%5.1f%s : %5.1f   (max %2d)",
                        param[i],ident[i],score-oldscore,max[i]);
                oldscore=score;
        }
        printf("\n\nScore %5.1f\n",score);
}


[ style 63.9 ]
```

THE OUTPUT

The output of the program suite, run for illustrative purposes against one of its own programs, is

```
Style analysis of style.detab.c

  TC   TL   MO   LC   BL   CL    NB    IN   RW   ID   IM   NL   IF   DF   JL
 706   43    3   25   11    7   418   168   10   13  120 4.62    1    2    (

16.7  characters per line      9.0   (max  9)
16.3% comment lines           12.0   (max 12)
23.8% indentation             11.8   (max 12)
30.6% blank lines              9.8   (max 11)
 4.8  spaces per line          8.0   (max  8)
 8.3  module length           10.8   (max 15)
10.0  reserved words           3.0   (max  6)
 4.6  identifier length        8.6   (max 14)
```

(*table continued overleaf*)

```
  0.0  gotos                      0.0    (max -20)
  1.0  include files              1.7    (max  5)
 15.4% defines                    8.0    (max  8)

Score  82.7
```

Appendix 4: Screen Characteristics

adm5.h

```
/* These are terminal control commands for an ADM5. */
/*            (Lear Siegler, 1981)                  */

#define HOME printf("\036")
#define CLEAR printf("\033Y")
#define CURSOR(l,p) printf("\033=%c%c", 31+l, 31+p)
```

vt100.h

```
/* These are terminal control commands for a VT100. */
/* They are but a few of the many available for     */
/* this device. The command names are chosen to     */
/* attempt to convey their purpose. The cursor      */
/* positioning command CURSOR is unusual for this   */
/* device, in that it requires an ascii string to   */
/* represent the positioning digits.                */
/* HOME is 1,1 .      (DEC, 1979)                   */

#define HOME printf("\033[H")

#define CLEARDOWN printf("\033[J")
#define CLEAR2EOL printf("\033[K")
#define CLEARSCRN printf("\033[2J")
#define CLEARLINE printf("\033[2K")
#define BOLD printf("\033[1m")
#define ULINE printf("\033[4m")
#define BLINK printf("\033[5m")
```

(continued overleaf)

```
#define REVERSE printf("\033[7m")
#define CANCEL printf("\033[0m")

#define CURSOR(l,p) printf("\033[%c%c;%c%cH",48+l/10,48+l%10,48+p/10,48+p%10)
```

Appendix 5: Tabulated and Listed Information

ALPHABETIC LIST OF C RESERVED WORDS

auto	storage class specifier
break	statement
case	statement prefix within a switch statement
char	type specifier
continue	statement
default	statement prefix within a switch statement
do	statement
double	type specifier
else	statement
entry	(reserved for future use)
enum	type specifier
extern	storage class specifier
float	type specifier
for	statement
goto	statement
if	statement
int	type specifier
long	type specifier
register	storage class specifier
return	statement
short	type specifier
sizeof	unary operator
static	storage class specifier
struct	type specifier
switch	statement
typedef	storage class specifier
union	type specifier
unsigned	type specifier
void	type specifier
while	statement

Use of any of these reserved words as identifiers will cause syntax errors. The ease with which such errors can be related to the source of the problem will depend on the particular implementation of C.

C ARITHMETIC OPERATORS

Operator	Name	Associativity	RatC
( )	parentheses	left to right	hier11
[ ]	brackets		hier11
–>	pointer		no
.	dot		no
++	increment	right to left	hier10
--	decrement		hier10
(type)	cast		no
*	contents of		hier10
&	address of		hier10
-	unary minus		hier10
~	one's complement		no
!	logical NOT		no
sizeof	size of		no
*	multiply	left to right	hier9
/	divide		hier9
%	modulus		hier9
+	plus	left to right	hier8
-	minus		hier8
>>	shift right	left to right	hier7
<<	shift left		hier7
>	greater than	left to right	hier6
>=	greater than or equal		hier6
<=	less than or equal		hier6
<	less than		hier6
==	equal	left to right	hier5
!=	not equal		hier5
&	bitwise AND	left to right	hier4
^	bitwise exclusive OR	left to right	hier3
|	bitwise inclusive OR	left to right	hier2
&&	logical AND	left to right	no
||	logical OR	left to right	no
?:	conditional	right to left	no

=	equals	right to left	hier1
+=	plus equals		hier1
-=	minus equals		hier1
*=	multiply equals		hier1
/=	divide equals		hier1
%=	modulus equals		hier1
>>=	shift right equals		hier1
<<=	shift left equals		hier1
&=	and equals		hier1
^=	exclusive or equals		hier1
|=	inclusive or equals		hier1
,	comma	left to right	no

C BASIC DATA TYPES

C supports the following basic data types.

char	A character variable holds any character from the available character set represented as the appropriate integer character code.
int	Up to three sizes of integers may be available: ***short int***, ***int*** and ***long int***. ***int*** represents the normal size of integer; ***short int***, if supported, will be no bigger than ***int***; ***long int***, if supported, will be no smaller than ***int***. Integers may be treated as signed, which is the default, or unsigned: ***unsigned short int***, ***unsigned int*** or ***unsigned long int***, where the word ***int*** is optional.
float	Represents a single-precision floating point number.
double	Represents a double-precision floating point number.
enum	Enumerated type, which may take any of a defined set of values.

ASCII CHARACTER SET (OCTAL)

000 nul ↑@	020 dle ↑P	040 sp	060 0	100 @	120 P	140 `	160 p
001 soh ↑A	021 dc1 ↑Q	041 !	061 1	101 A	121 Q	141 a	161 q
002 stx ↑B	022 dc2 ↑R	042 "	062 2	102 B	122 R	142 b	162 r
003 etx ↑C	023 dc3 ↑S	043 #	063 3	103 C	123 S	143 c	163 s
004 eot ↑D	024 dc4 ↑T	044 $	064 4	104 D	124 T	144 d	164 t
005 enq ↑E	025 nak ↑U	045 %	065 5	105 E	125 U	145 e	165 u
006 ack ↑F	026 syn ↑V	046 &	066 6	106 F	126 V	146 f	166 v
007 bel ↑G	027 etb ↑W	047 '	067 7	107 G	127 W	147 g	167 w
010 bs ↑H	030 can ↑X	050 (	070 8	110 H	130 X	150 h	170 x
011 ht ↑I	031 em ↑Y	051)	071 9	111 I	131 Y	151 i	171 y
012 nl ↑J	032 sub ↑Z	052 *	072 :	112 J	132 Z	152 j	172 z
013 vt ↑K	033 esc	053 +	073 ;	113 K	133 [	153 k	173 {
014 np ↑L	034 fs	054 ,	074 <	114 L	134 \	154 l	174 |
015 cr ↑M	035 gs	055 -	075 =	115 M	135]	155 m	175 }
016 so ↑N	036 rs	056 .	076 >	116 N	136 ^	156 n	176 ~
017 si ↑O	037 us	057 /	077 ?	117 O	137 _	157 o	177 del

ASCII CHARACTER SET (DECIMAL)

000 00 nul ↑@	016 10 dle ↑P	032 20 sp	048 30 0	064 40 @	080 50 P	096 60 `	112 70 p
001 01 soh ↑A	017 11 dc1 ↑Q	033 21 !	049 31 1	065 41 A	081 51 Q	097 61 a	113 71 q
002 02 stx ↑B	018 12 dc2 ↑R	034 22 "	050 32 2	066 42 B	082 52 R	098 62 b	114 72 r
003 03 etx ↑C	019 13 dc3 ↑S	035 23 #	051 33 3	067 43 C	083 53 S	099 63 c	115 73 s
004 04 eot ↑D	020 14 dc4 ↑T	036 24 $	052 34 4	068 44 D	084 54 T	100 64 d	116 74 t
005 05 enq ↑E	021 15 nak ↑U	037 25 %	053 35 5	069 45 E	085 55 U	101 65 e	117 75 u
006 06 ack ↑F	022 16 syn ↑V	038 26 &	054 36 6	070 46 F	086 56 V	102 66 f	118 76 v
007 07 bel ↑G	023 17 etb ↑W	039 27 '	055 37 7	071 47 G	087 57 W	103 67 g	119 77 w
008 08 bs ↑H	024 18 can ↑X	040 28 (	056 38 8	072 48 H	088 58 X	104 68 h	120 78 x
009 09 ht ↑I	025 19 em ↑Y	041 29)	057 39 9	073 49 I	089 59 Y	105 69 i	121 79 y
010 0A nl ↑J	026 1A sub ↑Z	042 2A *	058 3A :	074 4A J	090 5A Z	106 6A j	122 7A z
011 0B vt ↑K	027 1B esc	043 2B +	059 3B ;	075 4B K	091 5B [	107 6B k	123 7B {
012 0C np ↑L	028 1C fs	044 2C ,	060 3C <	076 4C L	092 5C \	108 6C l	124 7C |
013 0D cr ↑M	029 1D gs	045 2D -	061 3D =	077 4D M	093 5D]	109 6D m	125 7D }
014 0E so ↑N	030 1E rs	046 2E .	062 3E >	078 4E N	094 5E ^	110 6E n	126 7E ~
015 0F si ↑O	031 1F us	047 2F /	063 3F ?	079 4F O	095 5F _	111 6F o	127 7F del

ESCAPE CHARACTERS

The backslash character is used to construct 'escape sequences'. That is, it enables the user to represent certain non-printing characters by a pair of characters, backslash and one other. The characters represented in this way are

\b	backspace	BS
\f	form feed	FF
\n	newline	NL
\r	carriage return	CR
\t	horizontal tab	HT
\'	single quote	'
\\	backslash	\

A digit string of no more than three digits may also follow a backslash. This digit string is taken to be the octal representation of the required character in the underlying character set. For example, we may use

\0	null character	NUL
\7	bell	BEL
\177	rubout	DEL

Escape sequences such as those illustrated above may be used in strings, particularly control strings

```
printf("\t result = \n");
```

and also as character constants.

```
bell = '\7'
```

CONVERSION CHARACTERS FOR OUTPUT

Conversion characters	*Argument type*	*Comment*
c	char	Single character
d	int	Signed (if negative) decimal
ld or D	long	Signed (if negative) decimal
u	int	Unsigned decimal
lu or U	long	Unsigned decimal
o	int	Unsigned octal, zs
lo or O	long	Unsigned octal, zs

x	int	Unsigned hexadecimal, zs
lx or X	long	Unsigned hexadecimal, zs (zs . . . zero suppressed)
f	float or double	Decimal notation
e	float or double	Scientific notation
g	float or double	Shortest of %e, %f
s	string	

Any invalid conversion character is printed!

These conversion characters may be used in the control string of the function *printf*, and its variants *fprintf*, and *sprintf*. Examples of their use may be found in table 3.2.

CONVERSION CHARACTERS FOR INPUT

Conversion characters	*Argument type*
c	Pointer to char
d	Pointer to int
hd	Pointer to short
ld or D	Pointer to long
o	Pointer to int
ho	Pointer to short
lo or O	Pointer to long
x	Pointer to int
hx	Pointer to short
lx or X	Pointer to long
f	Pointer to float
lf or F	Pointer to double
e	Pointer to float
le or E	Pointer to double
s	Pointer to array of char

These conversion characters are for use in the control string of the function *scanf*, and its variants *fscanf*, and *sscanf*. Examples of their use are given in example 3.1.

INPUT-OUTPUT FUNCTIONS

Some or all of the functions below may be available in your implementation of C. Where appropriate we assume

arglist	is one or more arguments
cstring	is a control string
mstring	is the mode of a file, "r", "w", or "a"
fptr	is a pointer to a file

Functions that do not use files

getchar getchar( )

Read a character from the standard input. EOF is returned on end of file or when an error occurs.

putchar putchar(ch)

Write the character 'ch' to standard output, and return the character written.

printf printf(cstring, arglist)

Formatted print to standard output.

scanf scanf(cstring, arglist)

Read formatted from standard input. The function returns the number of arguments to which an assignment was made, or EOF, or NULL if the input did not match the first item of the control string.

gets char *gets(string)

Reads from the standard input a string, which is terminated by a newline character, into 'string'. *gets* returns its argument with the terminating newline character replaced by the null character.

puts puts(string)

Copies the string 'string' to standard output and appends a newline character.

sprintf sprintf(string, cstring, arglist)

Formatted write to the string 'string'.

sscanf sscanf(string, cstring, arglist)

Formatted read from the string 'string'.

Functions using files

fopen FILE * fopen(string, mstring)

Open the file with name 'string' in mode 'mstring'. The function returns as a result either a pointer to a file or NULL if the attempt to open was unsuccessful.

getc int getc(fptr)

Returns the next character from the file 'fptr'. EOF is returned on end of file or when an error occurs.

putc putc(ch, fptr)

Writes the character 'ch' to the file 'fptr'. EOF is returned on error, otherwise the character 'ch' is returned.

fgets char * fgets(string, n, fptr)

Reads into 'string' no more than n − 1 characters from 'fptr'. The read terminates upon finding a newline character, which is stored in 'string' followed by NULL. The function returns NULL on end of file or error, otherwise the first argument is returned.

fputs fputs(string, fptr)

Writes 'string' to 'fptr' with no newline appended.

fprintf fprintf(fptr, cstring, arglist)

Formatted write to 'fptr'.

fscanf fscanf(fptr, cstring, arglist)

Read formatted from 'fptr'.

fflush fflush(fptr)

Flush the output buffer of file 'fptr'.

ferror ferror(fptr)

Returns non-zero if an error has occurred while reading or writing 'fptr'. The error indication persists until the file is closed.

feof feof(fptr)

Returns non-zero when end of file has been reached on 'fptr'.

fclose fclose(fptr)

Any buffers associated with 'fptr' are emptied and the file is closed.

FUNCTIONS FOR STRING OPERATIONS

In what follows we assume the following declarations

```
char ch;
char *string1, *string2;
```

Some of the following functions may be available for string operations.

strcat — char *strcat(string1, string2)

Append a copy of 'string2' to the end of 'string1'. *strcat* returns a pointer to the result.

strcmp — strcmp(string1, string2)

Returns an argument less than, equal to, or greater than zero according as 'string1' is lexicographically less than, equal to, or greater than 'string2'.

strcpy — char *strcpy(string1, string2)

Copies 'string2' to 'string1' and terminates when the null character has been moved.

strlen — strlen(string1)

Returns the number of non-null characters in 'string1'.

strchr — char *strchr(string1, ch)

Returns a pointer to the first occurrence of character 'ch' in 'string1'.

strrchr — char *strrchr(string1, ch)

Returns a pointer to the last occurrence of character 'ch' in 'string1'.

strtok — char *strtok(string1, string2)

Returns a pointer to the next occurrence of one of the characters from 'string2' (the 'separators') in 'string1' (the 'token'), and writes a NULL in place of the separator. Subsequent calls with a NULL first argument step through the same string until no more tokens remain, at which time a NULL is returned.

The following variants limit their operation to the first 'n' characters of the string.

strncat — char *strncat(string1, string2, n)

strncmp — char *strncmp(string1, string2, n)

strncpy — char *strncpy(string1, string2, n)

MATHEMATICAL FUNCTIONS

While not normally considered to be a 'general-purpose' programming language, C implementations will usually offer a range of mathematical functions. Some or all of the following functions, in which 'x' and 'y' are of type *double*, might appear in an appropriate library file.

sqrt double sqrt(x)
Returns the square root of 'x'.

pow double pow(x, y)
Returns 'x' to the power 'y'.

fabs double fabs(x)
Returns the absolute value of 'x'.

ceil double ceil(x)
Returns the smallest integer not less than 'x'.

floor double floor(x)
Returns the largest integer not greater than 'x'.

exp double exp(x)
Returns the exponential function of 'x'.

log double log(x)
Returns the natural logarithm of 'x'.

log10 double log10(x)
Returns the logarithm base 10 of 'x'.

sinh double sinh(x)
Returns the hyperbolic function sinh of 'x'.

cosh double cosh(x)
Returns the hyperbolic function cosh of 'x'.

tanh double tanh(x)
Returns the hyperbolic function tanh of 'x'.

sin double sin(x)
Returns sine of 'x' when 'x' is in radians.

cos double cos(x)

Returns cosine of 'x' when 'x' is in radians.

asin double asin(x)

Returns, in radians, the arc sine of 'x'.

acos double acos(x)

Returns, in radians, the arc cosine of 'x'.

atan double atan(x)

Returns, in radians, the arc tangent of 'x' in the range −pi/2 to pi/2.

atan2 double atan2(x, y)

Returns, in radians, the arc tangent of 'x'/'y' in the range −pi to pi.

Appendix 6: Syntax Diagrams for C

In attempting to teach, or to use, a programming language, it is extremely helpful to have access to a concise specification of the syntax of the language, perhaps in the form of syntax diagrams. Unfortunately, it is not yet possible to provide a full and accurate description of the syntax of C in this fashion. This, it is suggested (Fitzhorn and Johnson, 1982), is because 'the language's syntax has never experienced a period of rigorous definition and design.' Early indications of this appear in Kernighan and Ritchie (1978) where it is admitted that 'the summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.' This, coupled with the later observation that C is an evolving language, gives an honest, if disappointing indication of why the C syntax is not so rigorously well-defined as might be expected. Authors and teachers are faced with a clear choice of either saying little or nothing about the definition of C syntax, or of presenting a syntax that might be considered erroneous. We have opted for presenting syntax diagrams that use the description of C syntax given by Fitzhorn and Johnson (1982), who acknowledge the critique by Anderson (1980). Readers are therefore cautioned to treat the following as an honest attempt to describe a useful syntax for C, rather than as a definitive syntax of C.

In the syntax diagrams that follow, upper case symbols are C keywords whereas lower case symbols are the names of syntactic elements. Three syntactic elements are not defined in the diagrams; their definitions are more concisely, if less formally, given here.

ident — An identifier is any sequence of upper case letters, lower case letters, digits or underscores. The sequence must start with a letter or an underscore.

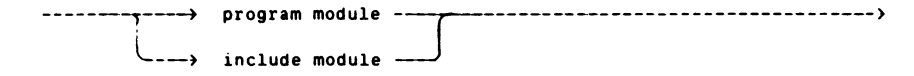
constant — A constant is a character constant, an integer constant or a floating constant. A character constant is a character enclosed by single quote marks. An integer constant is a digit sequence, which may be in hexadecimal form (starting 0x or 0X), in octal form (starting with 0), or in decimal form (starting with a non-zero digit). A floating constant is a digit sequence which should contain a decimal point and may

contain an exponent indicator (e, or E) followed by a signed or unsigned exponent.

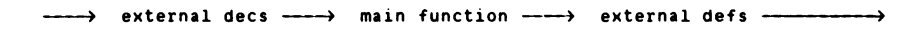
string — A string is any sequence of characters enclosed by double quote marks (").

(a) PROGRAM DEFINITION

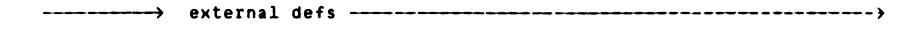
1. C program



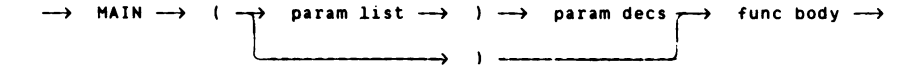
2. program module



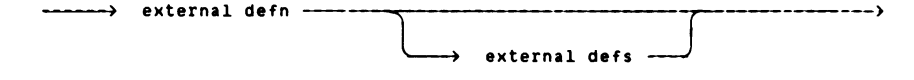
3. include module



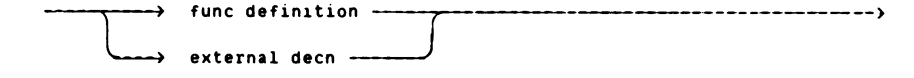
4. main function



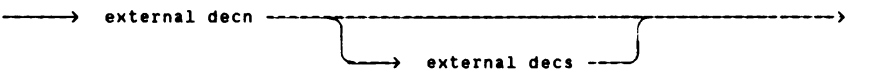
5. external defs



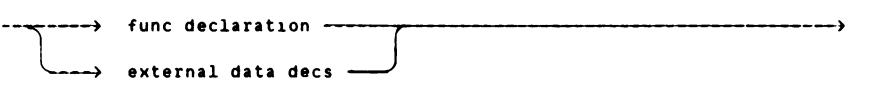
6. external defn



7. external decs

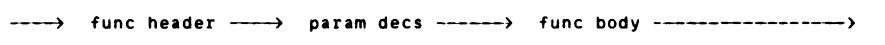


8. external decn



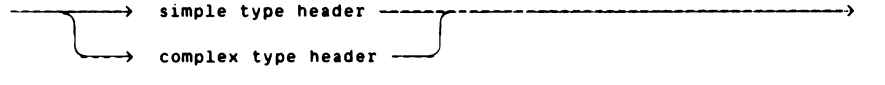
(b) FUNCTION DEFINITIONS

9. func definition

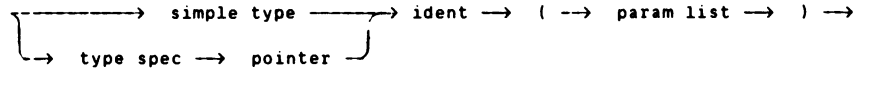


10. func header

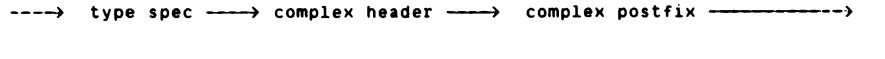
11. func type header



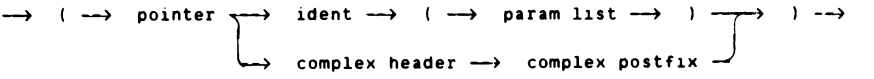
12. simple type header



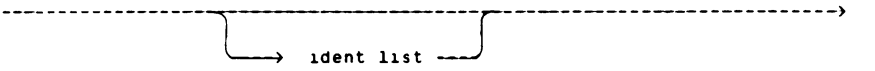
13. complex type header



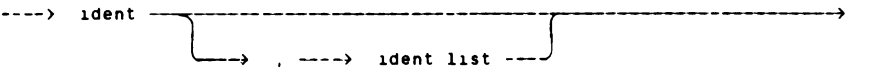
14. complex header



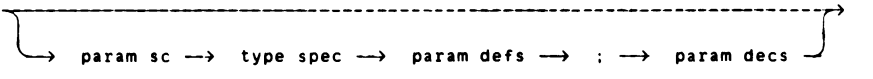
15. param list



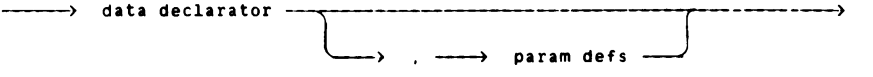
16. ident list



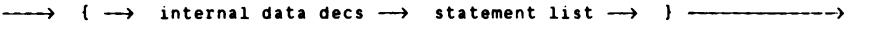
17. param decs



18. param defs

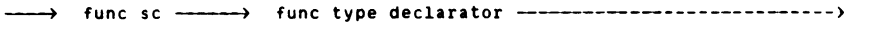


19. func body

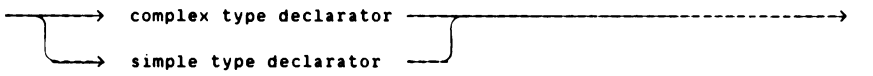


(c) FUNCTION DECLARATIONS

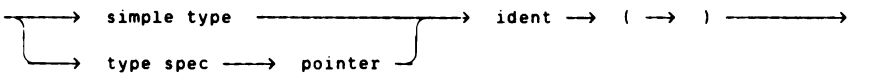
20. func declaration



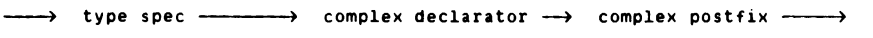
21. func type declarator



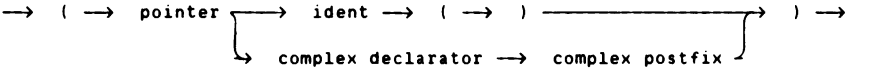
22. simple type declarator



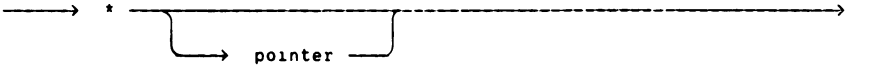
23. complex type declarator



24. complex declarator

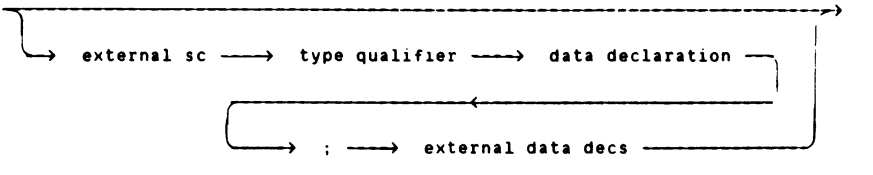


25. pointer

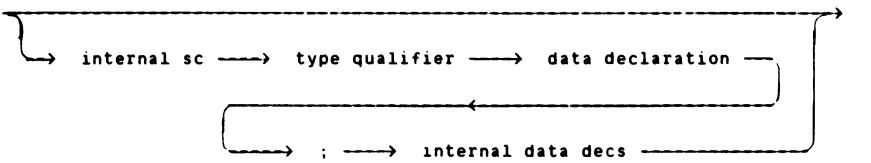


(d) DATA DECLARATIONS

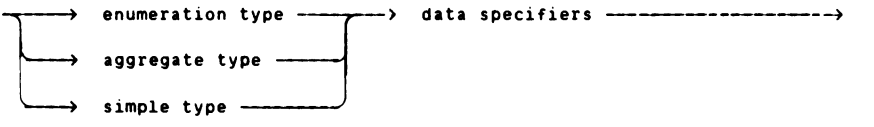
26. external data decs



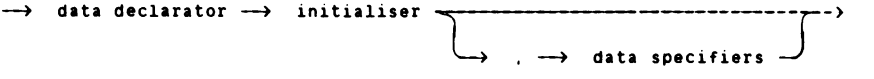
27. internal data decs



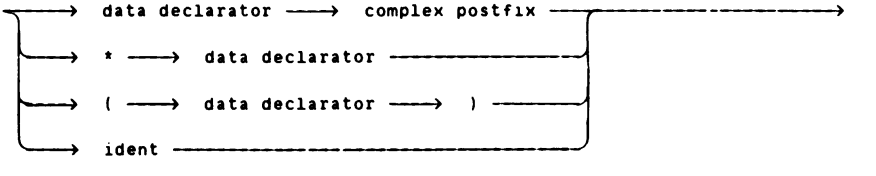
28. data declaration



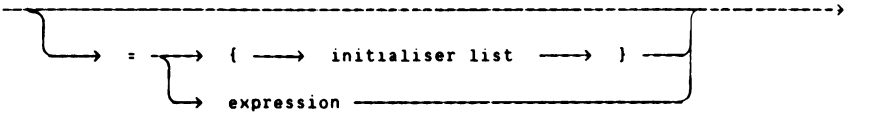
29. data specifiers



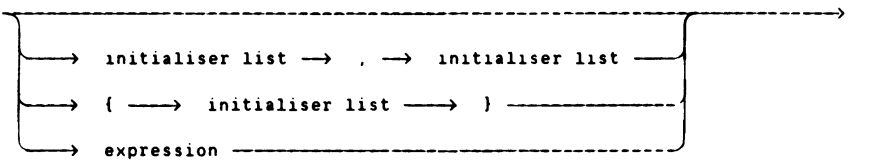
30. data declarator



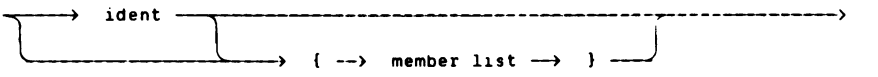
31. initialiser



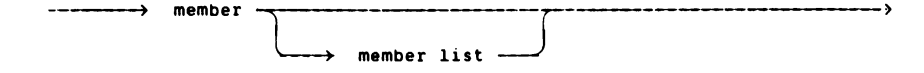
32. initialiser list



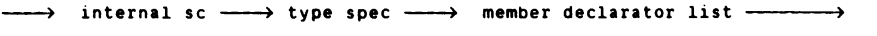
33. aggregate declarator



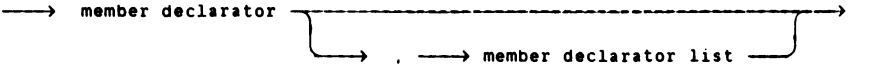
34. member list



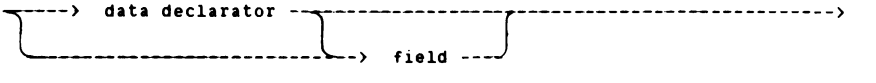
35. member



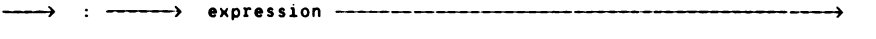
36. member declarator list



37. member declarator

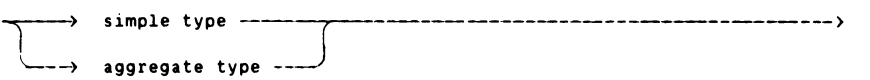


38. field

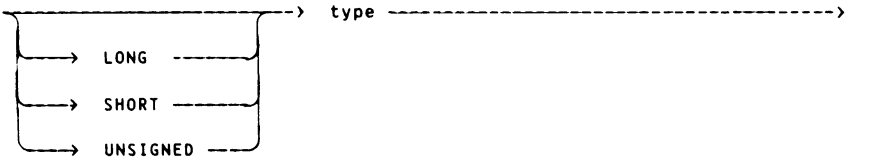


(e) TYPE ANALYSIS

39. type spec



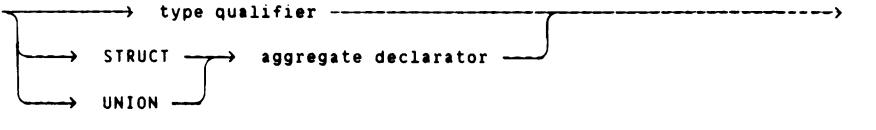
40. simple type



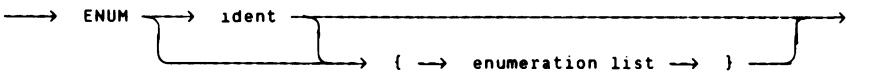
41. type

42. integer type

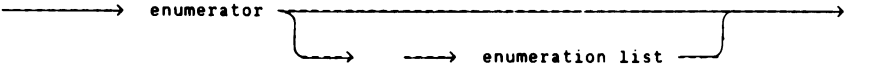
43. aggregate type



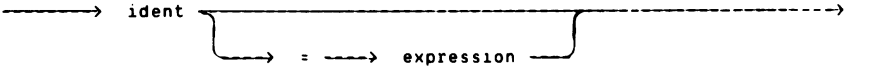
44. enumeration type



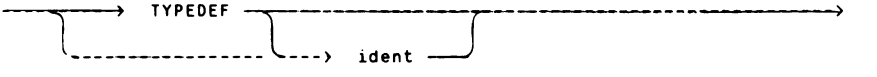
45. enumeration list



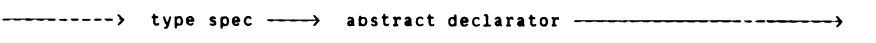
46. enumerator



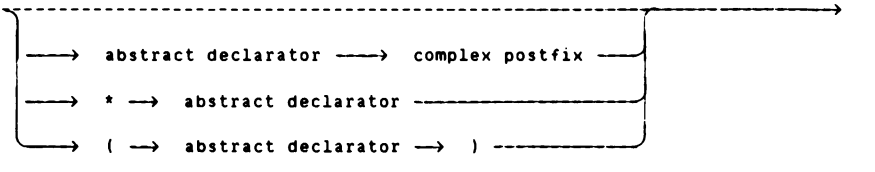
47. type qualifier



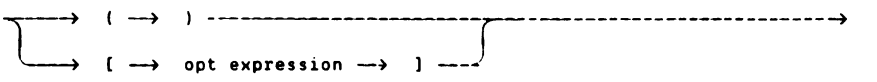
48. typename



49. abstract declarator

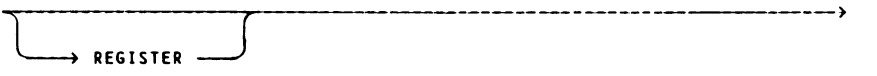


50. complex postfix



51. func sc

52. param sc

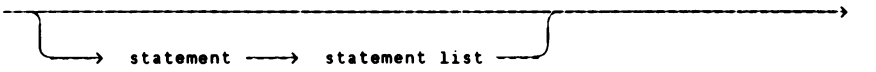


53. external sc

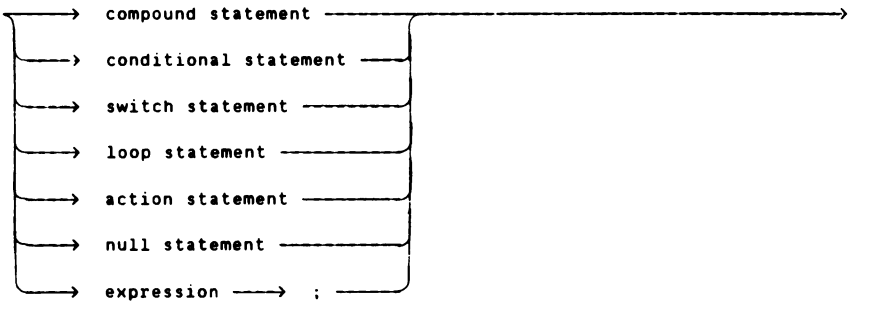
54. internal sc

(f) STATEMENTS

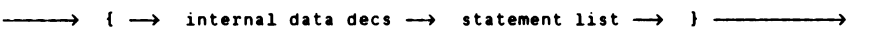
55. statement list



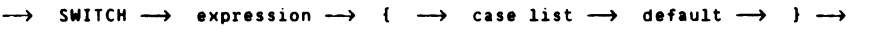
56. statement



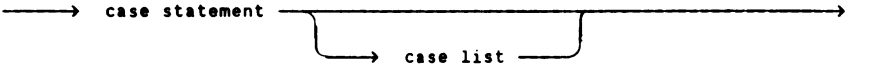
57. compound statement



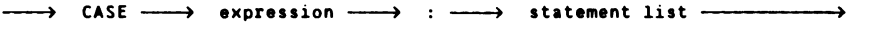
58. switch statement



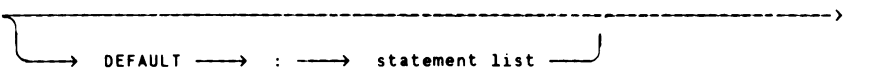
59. case list



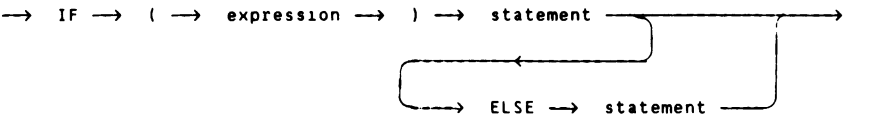
60. case statement



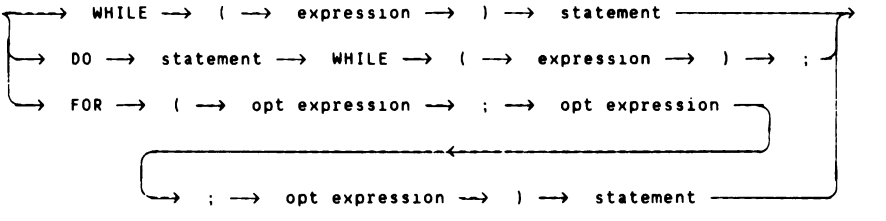
61. default



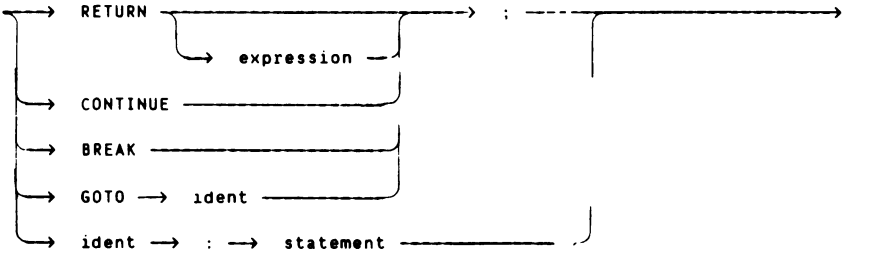
62. conditional statement



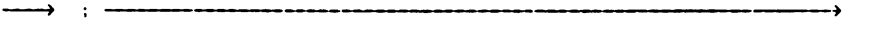
63. loop statement



64. action statement

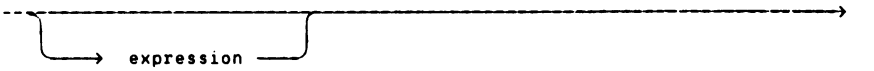


65. null statement

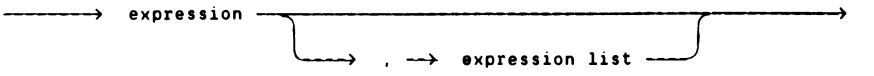


(g) EXPRESSIONS

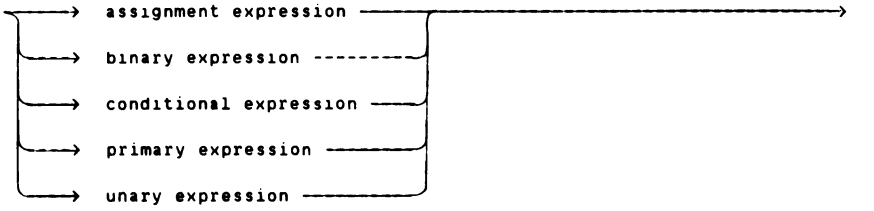
66. opt expression



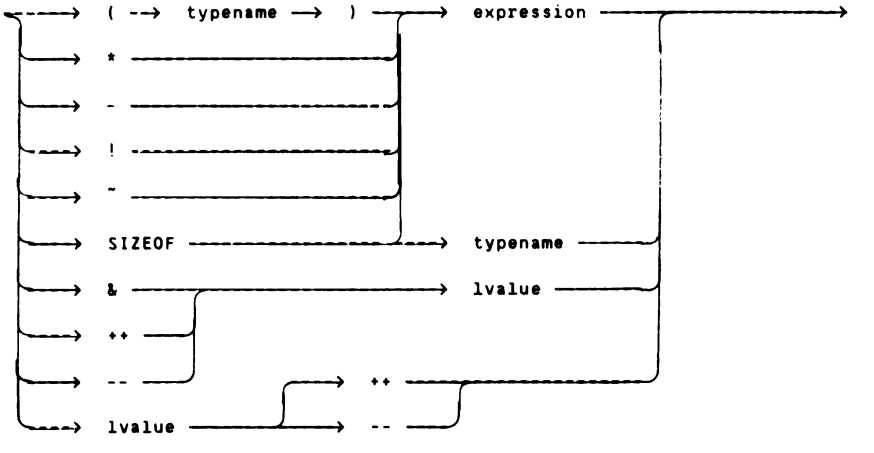
67. expression list



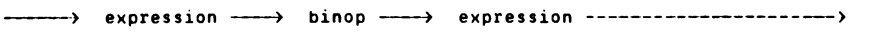
68. expression



69. unary expression

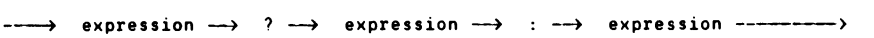


70. binary expression

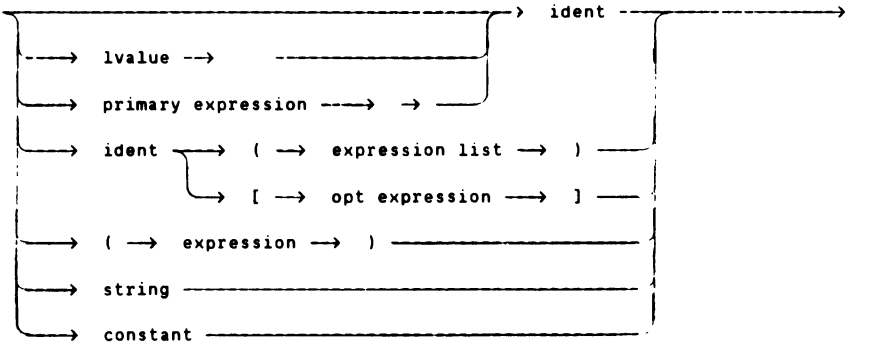


71. assignment expression

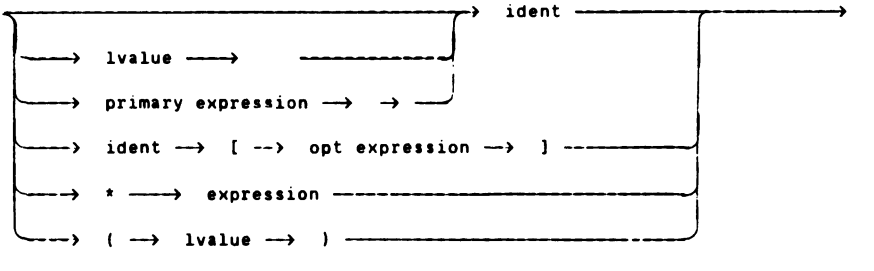
72. conditional expression



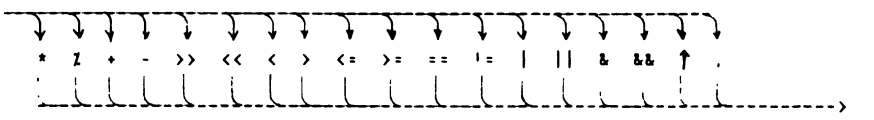
73. primary expression



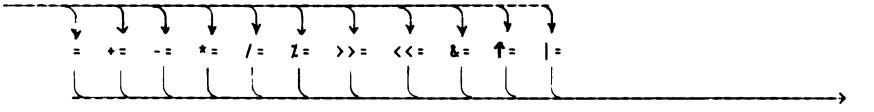
74. lvalue



75. binop



76. assignop



SYNTACTIC ELEMENTS IN ORDER OF DEFINITION

1. C program
2. program module
3. include module
4. main function
5. external defs
6. external defn
7. external decs
8. external decn
9. func definition
10. func header
11. func type header
12. simple type header
13. complex type header
14. complex header
15. param list
16. ident list
17. param decs
18. param defs
19. func body
20. func declaration
21. func type declarator
22. simple type declarator
23. complex type declarator
24. complex declarator
25. pointer
26. external data decs
27. internal data decs
28. data declaration
29. data specifiers
30. data declarator
31. initialiser
32. initialiser list
33. aggregate declarator
34. member list
35. member
36. member declarator list
37. member declarator
38. field
39. type spec
40. simple type
41. type
42. integer type
43. aggregate type
44. enumeration type
45. enumeration list
46. enumerator
47. type qualifier
48. typename
49. abstract declarator
50. complex postfix
51. func sc
52. param sc
53. external sc
54. internal sc
55. statement list
56. statement
57. compound statement
58. switch statement
59. case list
60. case statement
61. default
62. conditional statement
63. loop statement
64. action statement
65. null statement
66. opt expression
67. expression list
68. expression
69. unary expression
70. binary expression
71. assignment expression
72. conditional expression
73. primary expression
74. lvalue
75. binop
76. assignop

SYNTACTIC ELEMENTS IN ALPHABETIC ORDER

49. abstract declarator
64. action statement
33. aggregate declarator
43. aggregate type
71. assignment expression
76. assignop
70. binary expression
75. binop
1. C program
59. case list
60. case statement
24. complex declarator
14. complex header
50. complex postfix
23. complex type declarator
13. complex type header
57. compound statement
72. conditional expression
62. conditional statement
28. data declaration
30. data declarator
29. data specifiers
61. default
45. enumeration list
44. enumeration type
46. enumerator
68. expression
67. expression list
26. external data decs
8. external decn
7. external decs
6. external defn
5. external defs
53. external sc
38. field
19. func body
20. func declaration
9. func definition
10. func header
51. func sc
21. func type declarator
11. func type header
16. ident list
3. include module
31. initialiser
32. initialiser list
42. integer type
27. internal data decs
54. internal sc
63. loop statement
74. lvalue
4. main function
35. member
37. member declarator
36. member declarator list
34. member list
65. null statement
66. opt expression
17. param decs
18. param defs
15. param list
52. param sc
25. pointer
73. primary expression
2. program module
40. simple type
22. simple type declarator
12. simple type header
56. statement
55. statement list
58. switch statement
41. type
47. type qualifier
39. type spec
48. typename
69. unary expression

References

Anderson, B. (1980). 'Type syntax in the language C', *ACM Sigplan Notices*, **15**, No. 3 (March).

Bates, D. (1976). *Structured Programming* (State of the Art Report), Infotech, Maidenhead.

Berry, R. E. and Meekings, B. A. E. (1985). *Style analysis of C programs, Comm. ACM*, **28**, No 1, January.

Bleazard, G. B. (1976). *Program Design Methods*, National Computing Centre Publications, Manchester.

Bourne, S. R. (1982). *The UNIX System*, Addison-Wesley, London.

Cain, R. (1980a). 'A Small C compiler for the 8080's', *Dr Dobb's Journal*, No. 45 (May).

Cain, R. (1980b). 'A run time library for the Small C compiler', *Dr Dobb's Journal*, No. 48 (September).

Dahl, O-J., Dijkstra, E. W. and Hoare, C. A. R. (1972). *Structured Programming*, Academic Press, London.

DEC (1979). *VT100 User Guide*, Digital Equipment Corporation, Maynard, Massachusetts.

Feuer, A. F. (1982). *The C Puzzle Book*, Prentice-Hall, Englewood Cliffs, New Jersey.

Fitzhorn, P. A. and Johnson, G. R. (1982). 'C: Toward a concise syntactic description', *European UNIX User Group Newsletter*, **2**, No. 4 (winter).

Hall, J. (1982). 'A microprogrammed P-CODE interpreter for the Data General Eclipse s/130 minicomputer', *Software Practice and Experience*, **12**.

Hendrix, J. E. (1982). 'Small-C Compiler, v.2', *Dr Dobb's Journal*, No. 74 (December).

Kernighan, B. W. and Plauger, P. J. (1976). *Software Tools*, Addison-Wesley, Reading, Massachusetts.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey.

Knuth, D. E. (1973). *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Massachusetts.

Lear Siegler (1981). *ADM 5 Users' Reference Manual*, Lear Siegler, Inc., Anaheim, California.

Lewis, T. G. (1975). *Distribution Sampling for Computer Simulation*, D. C. Heath and Co., Lexington, Massachusetts.

Meekings, B. A. E. (1978). '"Random Number Generator" Algorith A-1', *Pascal News*, No. 12 (June).

Rees, M. J. (1982). 'Automatic assessment aids for Pascal programs', *ACM Sigplan Notices*, **17**, No. 10 (October).

Uspensky, J. V. and Heaslet, M. A. (1939). *Elementary Number Theory*, McGraw, New York.

Weems, C. (1978). 'Designing structured programs', *Byte* (August).

Wirth, N. (1976). *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey.

Index

! 59
!= 46, 60
#define 10, 99, 102
#else 99, 100
#endif 99, 100
#if 99, 101
#ifdef 99, 100
#ifndef 99, 101
#include 9, 99
#undef 99, 100
% 52
%= 56
& 23, 24, 58
&& 59
&= 56
*= 56
++ 46, 60
+= 56
, 61
-- 60
-= 56
/= 56
< 60
<< 58
<<= 56
<= 60
= 40
== 40, 60
> 60
>= 60
>> 58
>>= 56
?: 60

acos 212
actual parameter 22
address 23
alignment 76
allocation of storage 85
and operator 58
argc , argu 98
arguments 18
arithmetic operators 56, 202
arrays 65–75
  as parameters 68
  character 69
  declarations 65
  multi-dimensional 67
  of pointers 71
ASCII 12, 204
asin 212
assignment
  multiple 56
  operator 40, 56
  suppression 30
atan 212
atan2 212
automatic 7, 8

backslash 27
backspace 28, 206
BASIC 23
bit fields 81
bitwise operators 57
braces 11, 38
break 43, 47
bulletproofing 113

carriage return 28, 206
case 42
cast 55
ceil 211
char 6, 54, 76, 203
character arrays 69
clarity 105
closing a file 34
comma operator 61
comment 11
conciseness 108
conditional compilation 100
conditional operator 60
conditional statement 38

constants
  hexadecimal 58
  octal 58
continue 50
conversion characters 18, 27, 30, 206, 207
conversion specifiers 26, 30
cos 211
cosh 211

data structures 89
data types 76, 203
decisions 38
declaration
  array 65
  structure 77
  union 78
decrement 60
default 43
defensive programming 113
define 10, 12, 99, 102
derived type 77
do 45, 48
double 54, 203

efficiency 110
else 38, 39, 99, 100
endif 99, 100
enum 80, 203
enumeration 80
EOF 31, 35
equal operator 60
equivalence operator 40
escape character 28, 206
escape sequence 28, 206
exit 82
exp 211
expressions 38, 54–64
  order of evaluation 114
extern 9
external 6, 8

fabs 211
false 39
fclose 34, 209
feof 209
ferror 209
fflush 209
fgets 33, 34, 209
field width 26
FILE 35
file I/O 33
float 6, 54, 76, 203
floor 211
fopen 34, 35, 36, 209
fopen file modes 36
for 45, 49, 51, 52
form feed 28, 206
formal parameter 18, 22
fprintf 34, 209
fputs 33, 34, 209
free 87
fscanf 34, 209
functions 5, 14

getc 33, 34, 209
getchar 33, 34, 208
gets 208
goto 52
greater than operator 60
greater than or equal to operator 60

hexadecimal constants 58
hypothetical machine 118

identifier 6
if 38, 39, 99, 101
ifdef 99, 100
ifndef 99, 101
include 9, 10, 99
inclusive-or operator 58
increment 60
initialisation 46, 79
input 26, 29
int 6, 54, 76, 203
I/O function library 32

label 53
left shift operator 58
less than operator 60
less than or equal operator 60
linked list 87
lint 113
list, linked 87
log 211
log10 211
logical operators 59
long 54, 203
loops 45

macro 17
macro parameters 102
macro *vs* function 17
main 5
malloc 86
mathematical functions 211

memset 98
modulus operator 52
multi-dimensional arrays 67
multiple assignment 56
multiple conditions 41

newline 28, 206
not equal operator 60
NULL 35

octal constants 58
one's complement operator 58
opening a file 34
operator precedence 62-3
operators 54
  and 58
  arithmetic 56, 202
  assignment 56
  associativity 63, 114, 202
  bitwise 57
  comma 61
  conditional 60
  equal 60
  greater than 60
  greater than or equal to 60
  inclusive or 58
  left shift 58
  less than 60
  less than or equal 60
  logical 59
  not equal 60
  one's complement 58
  or 40
  relational 59
  right shift 58
output 26

parameter 18, 22, 23
parameters, macro 102
Pascal 5, 10, 23, 38, 59
pointer 23, 66, 84
  arithmetic 85
  arrays 71
  to structures 84
postfix ++ and -- 60
pow 211
precedence of operators 62-3
precision 27
prefix ++ and -- 60
preprocessor 9, 99
printf 11, 18, 26, 32, 34, 208
program structure 5
putc 33, 34, 209
putchar 33, 34, 208
puts 208

RatC 3, 6, 13, 23, 25, 32, 44, 45, 54, 59, 71, 75, 84, 110, 117
  cross-reference 175
  function list 184, 185
  implementation 123
  listing 130
  vs C 122
recursion 98
register 110
relational operators 59
reserved words 201
return 21, 22, 24, 47
right shift operator 58

scanf 29, 32, 34, 208
scope 8
screen characteristics 199
short 54, 203
sin 211
sinh 211
sprintf 32, 34, 208
sqrt 211
sscanf 32, 34, 208
static 7-9
stderr 33
stdin 33
stdio.h 11
stdout 32, 33
storage allocation 85
storage class
  automatic 7
  external 6
  static 7
strcat 210
strchr 210
strcmp 81, 210
strcpy 210
strings 32, 69, 70, 210
strlen 210
strncat 210
strncmp 210
strncpy 210
strrchr 210
strtok 98, 210
structure 76-9
  declaration 77
  pointers 84
  template 82
structure, program 5
style 1, 105, 186

style analysis 186
subscript 66
switch 42, 48
syntax diagrams 213

tab 28, 206
tanh 211
true 39
type, derived 77
type conversion 54
typedef 82, 83
types 76, 203

undef 99, 100
union 76–9
  declaration 78
UNIX 1, 2, 9, 11, 35, 98, 107, 110
unsigned 203

value parameter 23
VDU characteristics 12, 199
void 82

while 45–8
white space 29

\ 27
\' 28, 206
\b 28, 206
\f 28, 206
\n 28, 206
\r 28, 206
\t 28, 206
\\ 28, 206
^ 58
^= 56
| 58
|= 56
|| 40, 59
~ 58

Macmillan Computer Science Series

Consulting Editor:
Professor F. H. Sumner,
University of Manchester

In this revised edition, the coverage has been extended to include a complete treatment of the C programming language.

Discussions of data structures, the C preprocessor, together with additional operators and examples give the reader a clear view of advanced programming techniques.

As in the first edition, and in contrast to most other books on the C language, the authors reinforce the structured approach to programming by presenting the material in an order which parallels current thinking on the art of problem solving.

A source listing of a C compiler is included in an appendix.

The book will appeal to students and practitioners working in such areas as generic problem solving, algorithms, operating systems development, UNIX and compiler theory.

After postgraduate studies in computing at the University of Newcastle, **R.E. Berry** held the position of research associate there until 1970. He then became lecturer in the Department of Computing at the University of Lancaster, until his untimely death in 1985.

B.A.E. Meekings is a graduate of computing from the University of Lancaster, where he was also a lecturer until 1983 when he moved to the USA. He is now a senior analyst with Goldman Sachs & Co. in New Jersey.

M.D. Soren obtained a degree in physics and mathematics at Johns Hopkins University and studied physics in graduate school at the University of Maryland. Since 1984 he has been president of Soren Associates Inc., a New Jersey-based consultants specialising in UNIX and C.

ISBN 0-333-46529-6